

A Framework for Managing the Evolution of Business Protocols in Web Services

Seung Hwan Ryu¹, Régis Saint-Paul¹, Boualem Benatallah¹, Fabio Casati²

¹ School of Computer Science & Engineering
University of New South Wales
Sydney, Australia
{seungr, regiss, boualem}@cse.unsw.edu.au

² Intelligent Enterprise Technology Lab
HP Laboratories
Palo Alto, CA, 94304, USA
fabio.casati@hp.com

Abstract

Web services are loosely coupled software components that are published, discovered, and invoked across the Web. As the use of Web services grows, in order to correctly interact with the growing services, it is important to understand the business protocols that provide clients with the information on how to interact with services. In dynamic Web services environments, service providers need to constantly refine their business protocols in order to reflect the constraints and opportunities proposed by new applications, new business strategies, and new laws, or fix the problems found in the protocol definition. However, the effective management of such a protocol evolution raises challenging problems: one of the most challenging issues is to handle ongoing instances started with the old protocol when their protocols are changed.

We present a framework that supports service administrators in managing business protocol evolution by providing several features, such as a set of change operators allowing modifications of protocols and two types of change impact analyses automatically determining which ongoing instances can be migrated to the new version of a protocol. We have also implemented a database-backed GUI tool to manage the change process as an extension of our existing system.

Keywords: Web services, Business protocols, Evolution, Change impact analysis, Ongoing instances

1 Introduction

Web services, and more in general service-oriented architectures (SOAs) are quickly becoming the preferred choice for developing distributed applications and performing application integration within and across enterprises. The interface of Web services provided by service providers today is described using the Web Services Description Language (WSDL). Besides this, *business protocols* are rapidly gaining momentum and mindshare as a necessary part of the service description (Benatallah, Casati & Toumani 2005). A business protocol for a service is a specification of the possible conversations that a service can have with its clients (Alonso, Casati, Kuno & Machiraju 2004). Here, the conversation is a sequence of message exchanges to achieve a certain goal, e.g., booking flight tickets, where a message corresponds to an operation invocation of a Web service. The business protocols

play an important role in Web services environments since they provide developers with information on how to write clients (i.e., services) that can correctly interact with a given service, and they allow development tools and runtime middleware to deliver functionality that simplifies the service development life-cycle (e.g., automated code generation) (Benatallah, Casati & Toumani 2004).

In dynamic Web services environments, just as services evolve, the protocols associated with them also evolve. Namely, business protocols are rarely in stone since they could be changed for a variety of reasons, such as new applications, new business strategies, and new laws and regulations. Therefore, enterprises need to effectively manage the evolution of business protocols in order to meet the constraints and opportunities proposed by the changes. The problem of protocol evolution can be distinguished into two aspects:

- *Static protocol evolution* refers to the problem of modifying the protocol definition. To change a protocol, it is necessary to provide a set of change operations that allow the gradual modification of an existing protocol without the need of redefining it from scratch.
- *Dynamic protocol evolution* refers to the issue of changing a protocol while there are ongoing protocol instances started with an old protocol. There is a need for providing mechanisms for a protocol designer to handle the ongoing instances to meet the new requirements.

One of the most challenging issues in the modification of a protocol is to manage the protocol instances running according to the old protocol. The problem of handling the ongoing instances is motivated, for several reasons: (i) some protocols describe services that might be *of long duration* from several days to years, such as a citizenship application service, an insurance claim service, and a mortgage service, and so on. If all in-progress transactions are aborted to organize certain changes, all current clients would have to lose considerable amounts of work; (ii) there is a need for *minimizing* the disruption to current clients while making sure that the new protocol is applied; (iii) it is not adequate to manually manage ongoing instances, since protocols may be quite *complex* and there might be a huge number of active instances; (iv) there might be protocols for supporting services that are *of critical nature*, not to be stopped, such as in pharmaceutical and chemical industries. In this case, it is impossible to stop the services to meet the certain requirements resulted from some changes.

Due to the above reasons, simply aborting or cancelling all active instances are not adequate. Therefore, smarter approaches must be taken into account about how to deal with the *dynamic protocol evolution* problem.

In this paper, we propose a framework for effectively managing the evolution of business protocols used in Web Services. The contributions of the approach include:

- it provides the constraints that can be used as the features of a good protocol evolution management. We analyze and discuss the different issues that can arise in the protocol evolution. We identify several types of constraints that can occur when a protocol is changed, and show how the violation of such constraints can be detected and handled.
- it allows to automatically classify and group the instances, based on a variety of criteria. To do so, we analyze the impact of protocol changes on current active instances, through comparing protocols at the protocol level and examining properties of instances at the instance level. The classification of instances helps users in choosing a migration strategy suitable to each group.
- it provides a management tool for supporting the constraint violation detection and several analyses described above, which makes the evolution problem scalable. We use a relational database to store all the protocol information and ongoing instance related information, and query on them for the analyses.

In this paper we focus mainly on the *syntactic* perspective of protocol changes, e.g. the change of the sequence in which messages are exchanged within a protocol. We do not consider changes semantically. Namely, when comparing two protocols, we examine only the syntactic difference between them, rather than the semantic change.

The remainder of the paper is structured as follows. Section 2 describes business protocol models based on a finite state machine (FSM), migration strategies applicable to the classified instances, and issues relating to the protocol evolution. Then, we present protocol change operators modifying the protocol definition for the static protocol evolution. Section 3 introduces analyzing the impact of protocol changes on current ongoing instances with the two types of approaches. Section 4 describes the architecture of our system and the implementation of it. Finally, we discuss related work in Section 5 and conclude with a summary and directions for future work in Section 6.

2 Preliminaries

In what follows we present a protocol model developed in our previous work, and introduce the issues considered in protocol evolution and the protocol change operators for describing protocol changes.

2.1 Business Protocols Modeling

A business protocol specifies which message exchange sequences are supported by a service. By the business protocol, clients can know how to correctly interact with a service. Following our previous work (Benatallah et al. 2005), business protocols are expressed as a finite state machine (FSM). The reason for using FSM is because it is a well-known paradigm with established formal foundations that allows for analysis. It is also a simple and easy language to use and understand for non-expert users, and it is suitable for modeling reactive behaviors. FSM consists of states and transitions. The states represent the different phases that a service may go through

during its interaction with clients while transitions are triggered by messages sent by the clients to the service provider (hence, transitions are labeled with message). A message corresponds to the invocation of a service operation.

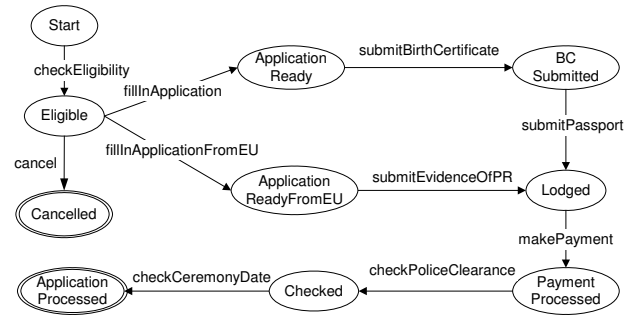


Figure 1: Business protocol P.I for an Australian citizenship application service

As an example, Figure 1 shows a graphical representation of a protocol, which describes the behavior of an Australian citizenship application service. A state is labeled with a logical name, such as *Eligible* and a transition with a message name. The protocol says that the citizenship application service is initially in the *Start* state, and that clients begin using the service by sending a *checkEligibility* message, upon which the service moves to the *Eligible* state. Clients from EU countries can proceed to state *Lodged* by filling in the application for EU people and submitting only the evidence of permanent residence, while clients from non EU countries proceed to the state *Lodged* by filling in the application, and submitting their birth certificate and passport. Then, after making payment and requesting a police clearance certificate, they check the ceremony date and complete the application service. In the following of the paper, this protocol will be denoted as P.I (initial version of the citizenship application service protocol).

We call a protocol instance any execution (occurrence) of a protocol. For example, in the protocol P.I, a particular citizenship application launched by a certain client represents an instance of the citizenship service protocol. Normally, several instances of the same protocol may be active in different states at the same time.

Formally, a business protocol is defined as follows:

Definition 2.1. (Business protocol)

A business protocol is a tuple $P = (\mathcal{S}, s_0, \mathcal{F}, \mathcal{M}, \mathcal{R})$ which consists of the following elements:

- \mathcal{S} is a finite set of states.
- $s_0 \in \mathcal{S}$ is the initial state.
- \mathcal{F} is a set of final states.
- \mathcal{M} is a finite set of messages. In our model, we assume that \mathcal{M} is a set of operation names.
- $\mathcal{R} \subseteq \mathcal{S}^2 \times \mathcal{M}$ is the transition relation. Each transition (s, s', m) identifies a source state s , a target state s' and a message m that is consumed during this transition.

2.2 Possible Migration Strategies

In this section, we explain the possible migration strategies applied to the ongoing instances. We detail these below:

- *Terminate*. The in-progress instances are allowed to continue to run according to the old protocol. This strategy is applicable when the provider can tolerate existing instances completing according to the old protocol.

- *Migration to new protocol.* The active instances are migrated to the new protocol without any conditions. The main goal of our work is to migrate instances to the new protocol as many as possible, with this strategy.
- *Migration to temporary protocol.* Temporary protocols may be defined for instances that are not migrateable to the new protocol. Its function is to make existing instances comply with the requirements causing the protocol changes from the old to the new protocol. Such a strategy is useful if the protocol modification is critically important but a cancel is considered to be disruptive.

2.3 Issues in Managing the Evolution of Business Protocols

Modifying a business protocol is challenging since the modifications may cause the violations of several types of constraints. We have identified three types of constraints that can be used as characteristics for a *good* protocol evolution management- one for static protocol evolution and the other two for dynamic protocol evolution:

- *Structural constraints* mean that the result of applying a protocol change operation to a legal protocol should be always a legal protocol, which refers to the protocol conforming to the protocol model. In addition, these constraints mean that changes should not cause any ongoing instances to end up in a situation such that it is not clear how to proceed (i.e., which messages are and are not allowed). One of important requirements in structural constraints is how to deal with the deletion of states. For example, if a protocol designer removes a state from a protocol, she must first check that all the remaining states are reachable from the initial state. Although our previous work (Skogsrud, Benatallah & Casati 2004) addressed the structural consistency of a security protocol, it can be applied for checking the structural consistency of a business protocol.
- *Correct interaction constraints* mean that the correct interaction of active instances with a given service should be assured after migration to the new protocol. The effects of applying change operations may result in the active instances failing of interaction with the service, since clients may not be prepared to interact with the new protocol, or in more significantly, they have already discovered and bound to the service described by the old one. This kind of constraint violation can occur only when active instances are migrated to a new protocol, since the migration can cause the fail of interaction. The fail of interaction does not occur when an ongoing instance is allowed to continue according to the old protocol, for there are no changes to the sequence of messages that it will take.
As an example, consider an applicant at state Eligible in the old protocol P.I (Figure 1). If the applicant's instance is migrated to state Eligible of the new protocol P.F (Figure 2), the violation of the correct interaction constraint might occur because the applicant might take the changed path (Eligible.fillInApplicationFromEU().submitBirthCertificate().Lodged) with which she could not get ready to interact.
- *Acceptable history constraints* mean that, after migration to the new protocol, each instance

should be seen as an instance of the new protocol in terms of history (trace) taken by the instance. In other words, the history of an instance must be accepted according to the new protocol. The unacceptable history means that the instance must have followed the changed parts of the old protocol and, hence, there is a need for taking some complementary actions to the instance, e.g., migrating it to a temporary protocol. Whether the history of an instance fits the protocol model is determined by considering the current state of the instance and actual path taken by it. In analogously to the previous constraint, this kind of constraint violation occurs when we migrate active instances to the new protocol.

For example, consider again the old and new protocols. We assume that there is an applicant currently in the state Lodged of the old protocol and this applicant's instance is migrated to the same state of the new one. If the applicant has followed the message sequence (Eligible.fillInApplicationFromEU().submitEvidenceOfPR().Lodged) in the old one, the migration causes the violation of acceptable history constraint since the message sequence (history) taken by her is not acceptable by the new protocol.

Because the structural constraints have been studied in prior research (Skogsrud et al. 2004), in this paper we focus on the other two constraints, *correct interaction constraint* and *acceptable history constraint*, since they are important in the protocol evolution for the following reasons: (i) the *correct interaction constraint* is necessary in order to guarantee the successful interaction between clients and services, which is agreement with the goal of business protocols; (ii) it is required to consider the *acceptable history constraint* in order to meet the new requirements causing the protocol change, which complies with the goal of protocol change.

In order to formally define the concepts related to the protocol evolution, we introduce the following symbols and definitions.

- Let $P = (\mathcal{S}, s_0, \mathcal{F}, \mathcal{M}, \mathcal{R})$ be an old business protocol and $P' = (\mathcal{S}', s'_0, \mathcal{F}', \mathcal{M}', \mathcal{R}')$ be a new business protocol.
- $State_P^I$ denotes the current state of an instance I in protocol P .
- Let an execution path $p = \langle s.m_0.m_1. \dots m_{k-2}.m_{k-1}.t \rangle$ be an alternating sequence of messages, from a state s to a state t , such that for $0 \leq i \leq k-1$, $m_i \in M$ and $s \in S \cup s_0$ and $t \in S \cup F$.
- $History_I^{s_0,t}$ denotes an execution path that starts from an initial state s_0 and ends at a state $t = State_P^I$, which was actually taken by instance I .
- $PathsFromStart_P^{s_0,t}$ denotes a set of execution paths that start from an initial state s_0 and end at a state $t = State_P^I$, $t \in S$ in protocol P .
- $PathsToCompletion_P^{t,f}$ denotes a set of execution paths that start from a state $t = State_P^I$, $t \in S$ and end at a final state $f \in F$ in protocol P .

Definition 2.2. (Correct interaction)

In the migration of instance I from protocol P to protocol P' , interactions that can be taken by the instance I in protocol P are correct in the context of protocol P' iff $PathsToCompletion_P^{t,f} \subseteq PathsToCompletion_{P'}^{t,f}$.

This definition states that, if the new protocol includes all the possible interactions that can occur between the instance’s current state and final states in the old protocol, the instance can correctly interact with the new protocol.

Definition 2.3. (Acceptable history)

In the migration of instance I from protocol P to protocol P' , history taken by the instance I in protocol P is acceptable in the context of protocol P' iff $History_I^{s_0,t} \in PathsFromStart_{P'}^{s_0,t}$.

This definition states that, if the actual history taken by the instance belongs to the set of possible paths from the initial state to the state corresponding to the instance’s current state in the new protocol, the history is acceptable in the context of the new protocol.

Definition 2.4. (Safe migration)

Migration of instance I from protocol P to protocol P' is safe iff the interactions of I satisfy the definition 2.2 and the history of I satisfies the definition 2.3.

2.4 Protocol Change Operators for Static Protocol Evolution

The evolution process begins by modifying the existing protocol. All instances starting in the future will be processed on the basis of the new protocol. To allow protocol changes, it is necessary to provide a set of change operators that can be applied to a protocol. It should be possible to go from any protocol to any protocol with these operators. We present four types of operators as follows:

- **AddTransition** (Message m , State s , State t): This operator adds a transition labeled with message m between source state s and target state t .
- **RemoveTransition** (Message m , State s , State t): This operator removes a transition labeled with message m between source state s and target state t .
- **AddState** (State r , State s , Message m): This operator adds a new state s in the new protocol as a successor of the state r . The transition with m is added from state r to state s .
- **RemoveState** (State s): This operator removes the state s from the protocol. Before s is removed, it should be checked that all target states of outgoing transitions from s can be reachable from the initial state even after the deletion of the state s . Also, after state s is removed, all in- and out-transitions of s are removed.

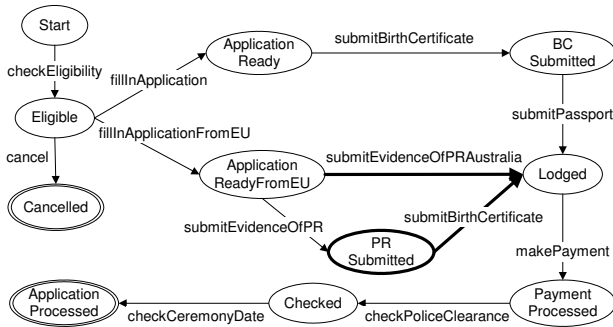


Figure 2: Changed protocol P.F for an Australian citizenship application service

Referring to the citizenship application protocol, suppose that there are some amendments in the law relating to the citizenship application as follows:

- The updated law proposes the requirement that, like non EU applicants, applicants from EU countries should submit the birth certificate as well as the evidence of permanent residence to lodge their application.
- Exceptionally, if applicants from EU countries were born in Australia, they can lodge the application without providing their birth certificate.

Therefore, the service protocol must be modified to meet the new requirements. The new protocol is depicted in Figure 2 (newly inserted parts are bold). This protocol will be denoted in the following as P.F (final version of the citizenship application protocol). In this protocol, a state and two transitions have been inserted. The final version P.F can be obtained with the following operators:

- AddState(ApplicationReadyFromEU, PRSubmitted, submitEvidenceOfPR)
- AddTransition(submitBirthCertificate, PRSubmitted, Lodged)
- AddTransition(submitEvidenceOfPRAustralia, ApplicationReadyFromEU, Lodged)

The first operator adds the new state PRSubmitted as a successor of state ApplicationReadyFromEU, while the last two instructions insert the transition submitBirthCertificate between PRSubmitted and Lodged and then add the transition submitEvidenceOfPRAustralia to be invoked from the applicants who were born in Australia. In order to achieve the desired result, the protocol designer needs to execute the change operators atomically and in the specified sequence.

3 Analyzing the Impact of Protocol Changes on Current Active Instances

The analysis of change impact can be done by examining the protocols themselves, or analyzing the properties of active instances (e.g., execution path taken by an instance). In this section, we introduce the two analysis approaches (i.e., *static* and *dynamic* analysis) and describe how to classify the active instances into the groups of instances *migrateable* to the new protocol using the result of the analysis.

3.1 Static Analysis at the Protocol Level

The static analysis is made at the protocol level (e.g., which parts of an old protocol are not affected by the changes). The purpose of this analysis is to examine the two protocols in order to identify whether a new protocol can replace an old one or which sub-protocols might cause the violations of the constraints, without individually looking at ongoing instances at the instance level. After this analysis, we can classify the active instances into groups of migrateable and non-migrateable instances. For example, if a new protocol can replace an old one, all the active instances begun under the old protocol can be migrated to the new one.

For the analysis, we present a *protocol replaceability* and three types of *sub-protocol analyses (SPAs)* between the two business protocols: *incorrect interaction causing SPA*, *unacceptable history causing SPA*, and *unaffected SPA*. These provide a foundation of analyzing the impact of protocol changes.

Protocol replaceability. To determine whether a new protocol (P.F) can be used instead of the old one (P.I), this analysis makes sure that the new protocol can support all the message sequences (conversations) which are provided by the old one. If the new one does, we say that the new protocol can replace the old one, which means that there is no problem about the migration of active instances because they will be able to run under the new one. It occurs in situations where the changes are parallelly additive, e.g., when a new transition is inserted between two certain states that have already another transition.

Unaffected SPA. If a new protocol cannot replace an old one, we need to classify active instances into several groups in order to choose a migration strategy appropriate for the groups of classified instances. The purpose of classification is to filter out instances migrateable to the new protocol as many as possible.

This class of analysis determines the sub-protocol that are *not* affected by the changes. The states within this sub-protocol have the same interaction and history paths in the old and new protocols, which means that the instances in the states followed the unaffected (normal) history path and will follow the normal interaction path. Thus, all the instances in the states of the sub-protocol identified by this kind of analysis can be migrated to the new protocol without generating any kind of constraint violations. To perform this analysis, we develop a function that takes as input two protocols (old and new protocols) and generates the states of the sub-protocol which is not affected by the changes, regardless of whether the changes are tolerable. This analysis is important as it is independent from the instances, so we do not need to repeat the analysis at the instance level.

For example, in Figure 1, the analysis function returns three states ApplicationReady, BCSubmitted and Cancelled, for the states have the same history and interaction paths in the old and new protocols. Thus, we can migrate all the instances in these states to the corresponding states of the new protocol.

Unacceptable History Causing SPA. After conducting the two analysis methods above, we need to determine whether the net result of a sequence of changes would cause any constraint violations. To do so, this class of analysis calculates the states in the sub-protocol of the old protocol, which might cause the unacceptable history in case that the active instances in the states are transferred to the new one.

A function doing this analysis takes as input two protocols and generates states of the sub-protocol located *after* the changed parts of the old protocol, since only the instances already past the changed parts might have the unacceptable history in the context of the new protocol. In other words, some of instances in the states of the sub-protocol might have followed the normal path (acceptable history in conformance with the new protocol) while the others might have taken the path affected by changes (unacceptable history). So, if we migrate all the active instances in the states to the new protocol, the migration could cause the violation of the *acceptable history constraint*. In our example, states Lodged, Payment-Processed, Checked and ApplicationProcessed belong to the sub-protocol.

Here, we can know that it is necessary to *further* examine the paths taken by individual instances in the states in order to exactly extract migrateable instances. This will be addressed in the dynamic analysis (in Section 4.2). However, this kind of analysis may help the protocol designer to choose a migration strategy, e.g., if the changes done to the old protocol are *tolerable*, regardless of the changes, the protocol

designer can migrate all instances to the new protocol.

Incorrect Interaction Causing SPA. This class of analysis identifies states of the sub-protocol located *before* the changed parts of the old protocol. Hence, the states in this sub-protocol have the changed execution paths to final states, which means that some of instances in the states would follow the normal path (correct interaction with the service) while the others would the path affected by changes (incorrect interaction with the service). Migrating the instances in the states of the sub-protocol directly to the new protocol could cause the violation of the *correct interaction constraint*, for the states have the changed execution paths in the new protocol. However, this kind of analysis is useful in choosing a migration strategy, e.g., if the changes are *tolerable* to the service provider, all active instances can be allowed to continue to run under the old protocol (i.e., *Terminate* strategy).

For example, this analysis generates the sub-protocol consisting of states Start, Eligible and ApplicationReadyFromEU.

The formal definitions related to the static analysis are below.

Definition 3.1. (Sub-Protocol)

Let $P = (S, s_0, \mathcal{F}, \mathcal{M}, \mathcal{R})$ be a business protocol. A sub-protocol is a tuple $SP = (S^{SP}, s_0^{SP}, \mathcal{F}^{SP}, \mathcal{M}^{SP}, \mathcal{R}^{SP})$ consisting of the following elements:

- $S^{SP} \subseteq S$ is the set of states of the sub-protocol SP .
- $s_0^{SP} \in S \cup s_0$ is the initial state of the sub-protocol SP .
- $\mathcal{F}^{SP} \subseteq S \cup \mathcal{F}$ is the set of final states of SP .
- $\mathcal{M}^{SP} \subseteq \mathcal{M}$ is the set of messages of SP .
- $\mathcal{R}^{SP} \subseteq \mathcal{R}$ is the transition relation of SP .

Note that, when calculating a sub-protocol, we are only interested in a set of states within the sub-protocol since identifying them based on some analysis enables us to decide how to handle all the active instances at the states. Hence, the below SPA definitions are related to identifying the states of a sub-protocol.

Definition 3.2. (Static Analysis Classes)

Let $P = (S, s_0, \mathcal{F}, \mathcal{M}, \mathcal{R})$ and $P' = (S', s'_0, \mathcal{F}', \mathcal{M}', \mathcal{R}')$ be two business protocols.

- **Protocol replaceability:** Let a complete execution path be a path that starts from an initial state and ending at a final state. A protocol P' can replace P iff P' supports all the complete execution paths that P supports.
- **Unacceptable history causing SPA:** An unacceptable history causing sub-protocol is a set of states t , with $t \in S \cap S'$ such that for $\exists h \in PathsFromStart_P^{s_0, t}$, $h \notin PathsFromStart_{P'}^{s'_0, t}$.
- **Incorrect interaction causing SPA:** An incorrect interaction causing sub-protocol is a set of states t , with $t \in S \cap S'$ such that for $\exists i \in PathsToCompletion_P^{t, f}$, $i \notin PathsToCompletion_{P'}^{t, f}$.
- **Unaffected SPA:** An unaffected sub-protocol is a set of states with $t \in S \cap S'$ such that for $\forall h \in PathsFromStart_P^{s_0, t}$, $\forall i \in PathsToCompletion_P^{t, f}$, then $h \in PathsFromStart_{P'}^{s'_0, t}$ and $i \in PathsToCompletion_{P'}^{t, f}$.

Given the old and new protocols, algorithm 1 computes the states of the sub-protocol of causing unacceptable history. For each state that exists commonly in the two protocols, the algorithm calls the procedure *GetExecutionPaths*(P, s) to get a set of execution paths from the initial state to the state in the old and new protocols (lines (3) to (4)). If any execution path from the initial state to a state of the old protocol does not belong to the set of execution paths from the initial state to the same state of the new one, the state is added to the variable *Candidates* (lines (5) to (9)). We have omitted the details of the other algorithms for space reasons.

Algorithm 1: Unacceptable history causing SPA

Input: $P = (S, s_0, \mathcal{F}, \mathcal{M}, \mathcal{R})$ and $P' = (S', s'_0, \mathcal{F}', \mathcal{M}', \mathcal{R}')$.
Output: A set of states.
begin
1: Let *Candidates* := { };
2: **foreach** $s \in S \cap S'$ **do**
3: *PathsFromStart* := *GetExecutionPaths*(P, s);
4: *PathsFromStart'* := *GetExecutionPaths*(P', s);
5: **foreach** $h \in \text{PathsFromStart}$ **do**
6: **if** $h \notin \text{PathsFromStart}'$ **then**
7: *Candidates* := *Candidates* \cup s ;
8: **break**;
9: **endfor**
10: **endfor**
11: **return** *Candidates*;
end

Algorithm 2: GetExecutionPaths

Input: protocol $P = (S, s_0, \mathcal{F}, \mathcal{M}, \mathcal{R})$ and state s
Output: a set of execution paths from s_0 to s .
begin
1: Let *executionPaths* := { };
2: Let *executionPath* := "";
3: **if** $s = s_0$ **then**
4: *executionPaths* := *executionPaths* \cup *executionPath*;
5: **else**
6: *parentStates* := parent states of s ;
7: *incomingMessages* := incoming messages of s ;
8: **foreach** $parentState \in parentStates$
9: and $message \in incomingMessages$ **do**
10: *parentPaths* := *getParentPaths*($parentState$);
11: **foreach** $parentPath \in parentPaths$ **do**
12: *executionPath* := $parentPath + "." + message$;
13: *executionPaths* := *executionPaths* \cup *executionPath*;
14: **endfor**
15: **endfor**
16: **return** *executionPaths*;
end

3.2 Dynamic Analysis at the Instance Level

To filter out migrateable instances, it is not sufficient to classify active instances based on the static analysis of protocols. Therefore, for categorizing active instances into migrateable instances at the instance level, we propose three additional analysis methods: *future path*, *past path*, and *instance property analysis*.

Future path analysis using a client protocol refers to the analysis of the expected path that an instance could take in the old protocol. Using the future path information, we can see that the client will never follow the changed path and there is no problem in migrating the client's instance to the same state in the new protocol as it were in the old one. We can identify the future path through considering the client protocol, or analyzing attributes of clients. In this paper, we assume that we can know the client's protocol, e.g., in case that the University of New South Wales purchases computers from Dell company, they

can know the partner's protocol each other. This dynamic analysis is used for further extracting the migrateable instances from the instances in the states of the sub-protocol generated by the *incorrect interaction SPA*.

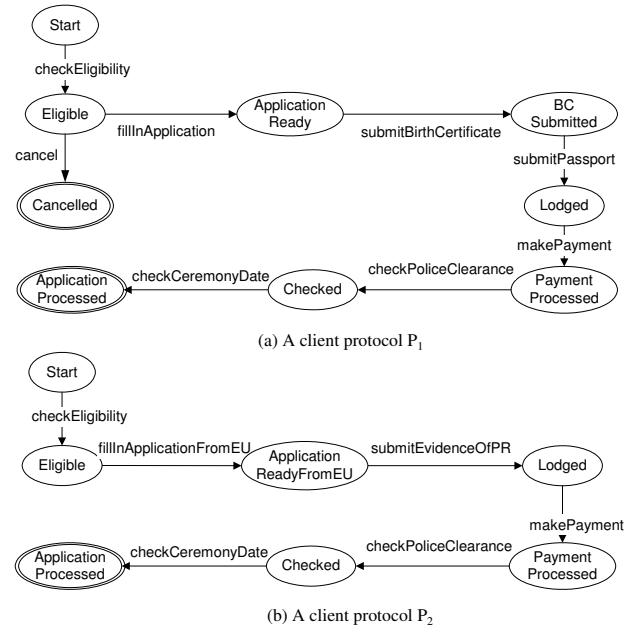


Figure 3: Client protocols interacting with the Australian citizenship application service P.I

As an example of using client protocols to identify the future path, consider the service protocol P.I depicted in Figure 1 and its client protocols of Figure 3. We assume that there are two clients at state *Eligible* in the protocol P.I, and one (client 1) behaves according to the client protocol P_1 of Figure 3 and the other (client 2) interacts with the given service according to the client protocol P_2 of Figure 3. In this case, the client 1's instance can be migrated to the state *Eligible* of the new protocol P.F, for the client will never take the changed path according to her protocol. The migration does not cause the violation of *correct interaction constraint*. In contrast, we cannot migrate the client 2's instance to the corresponding state of the protocol P.F since she will follow the changed path and fail to correctly interact with the given service, which means that the migration will generate the violation of the constraint.

Past path analysis using an actual history refers to the analysis of the actual path that an instance took. This analysis plays an important role in filtering out migrateable instances from the instances in the states of the sub-protocol identified by the *unacceptable history causing SPA*.

As an example of using the past path, consider the state *PaymentProcessed* of the old protocol in Figure 1. If all the instances in the state are migrated to the state *PaymentProcessed* of the new protocol, this causes the violation of *acceptable history constraint*, namely, a breach of the new law since it requires clients from EU countries to submit their birth certificate for the citizenship application. So, to filter out migrateable instances, there is a need for analyzing the actual past path taken by individual instances in the state. By doing this, the protocol designer knows that some of them followed the path 1 (Start.ApplicationReady.submitBirthCertificate(). BCSubmitted.PaymentProcessed) while the others followed the path 2 (Start.ApplicationReadyFromEU.submitEvidenceOfPR().Lodged.PaymentProcessed). In this case, only the instances followed the path 1 must be migrated to the state *PaymentProcessed*.

Instance property analysis. Another important information to be considered in the dynamic analysis is to examine properties of instances, e.g., application id, nationality, age, etc. The properties can be determined at the time of modeling business protocols or classifying active instances. This information is useful to further filter out migrateable instances from the non-filtered instances that will take the changed future path or took the changed past path.

For example, suppose that we choose a property, that is, whether a client was born in Australia. If we apply this information to the non-filtered instances in state `PaymentProcessed` that followed the changed path (i.e., path 2 in the previous example), the instances that meet the property (born in Australia) can be moved to the corresponding state of the new protocol, since the migration does not cause the violation of *acceptable history constraint* in the context of the new protocol.

Definition 3.3. (Dynamic Analysis Classes)

Let P and P' be two business protocols, and CP be a client protocol.

- **Future path analysis:** Let $CommonPaths^{t,f}(P, CP)$ be a set of common execution paths between P and CP , from a state $t=State_P^I$ to a final state $f \in F$. An instance I is migrateable to protocol P , with respect to client protocol CP , iff $\forall p \in CommonPaths^{t,f}(P, CP), p \in PathsToCompletion^{t,f}$.
- **Past path analysis:** An instance I is migrateable to protocol P , with respect to its history $History_I^{s_0,t}$, iff $History_I^{s_0,t}$ satisfies the definition 2.3.
- **Instance property analysis:** Let $Properties-Set$ be a set of instance properties specified by a protocol designer, and $InstanceProperties$ be a set of actual instance properties. An instance I is migrateable to protocol P , with respect to its property set $InstanceProperties$, iff $\forall p \in Properties-Set, p \in InstanceProperties$.

Figure 4 shows the classification hierarchy generated after the *static* and *dynamic* analysis. The grid circle means that all the instances in the circle can be transferred to the new protocol or allowed to run continuously under the old protocol while the half-grid circle means that only some of the instances can be migrated.

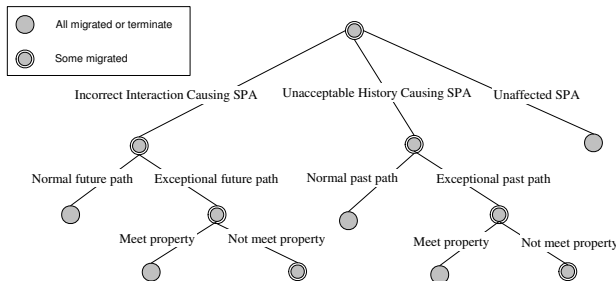


Figure 4: Classification hierarchy resulted from the static and dynamic analysis

3.3 Handling Instances Classified as Non Migrateable

How to deal with the instances that cannot be still filtered out by the static and dynamic analysis? In

many situations it is not adequate to abort the non-migrateable instances and restart them in the new protocol in order to achieve the business protocol goal according to the new requirements, for a huge amount of work already done might be lost if the clients have to start from the beginning, and also the clients cannot understand the new protocol and correctly interact with it. Alternatively, the protocol designer may prefer to make instance specific adaptation to achieve the goal. For this reason, we propose two approaches as follows:

- **Developing an adapter.** If a protocol designer defines a new protocol, mismatches between a client’s protocol and the newly defined one occur, which may cause the client to fail in the interaction with the service when her instance is migrated to the new protocol. In our previous work (Benatallah, Casati, Grigori, Motahari Nezhad & Toumani 2005), we proposed an approach for developing an adapter to resolve the differences that occur at the protocol level. Through the developed adapter, the client can continue to correctly interact with the new protocol, even when mismatches happen between the old and new protocols. In the work, we presented a taxonomy of possible mismatches and proposed a solution to tackle each kind of mismatch. We refer the interested reader to the work (Benatallah et al. 2005) for the detailed description.
- **Notice-Wait-Receive.** In some cases, it is not possible to develop the adapter to bridge the protocol differences. To handle the cases, we construct a temporary protocol whose purpose is to meet the new requirements without cancelling ongoing instances. Before migrating the instances to the temporary protocol, we notice clients of the protocol modification, and let them know about the temporary protocol with the time limit within the client sides are allowed to make their system adjustments (e.g., one week). And then, the service provider waits for the time period and receiving the clients’ replies. If the reply is that the client side successfully modified its protocol and system according to the temporary protocol, the service provider migrates the client’s instance to the temporary one. In some cases, clients can easily do the system adjustments through automatically generating the adapters and related source code using our adapter developing mechanisms.

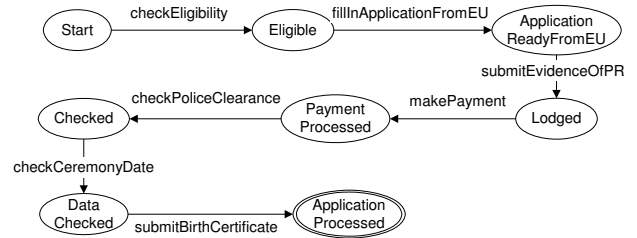


Figure 5: Example of a temporary protocol

This approach is viable, since (i) waiting for clients to make adjustments may much less disruptive than cancelling the work already done, in particular, long-duration service that might take several months; (ii) if the clients are *humans*, rather than *client machines*, they are much more flexible to behave according to the temporary protocol as there is no need for making adjustments.

As an example, consider the citizenship application protocol (Figure 1). If there are some non-migrateable instances in state *Checked* after performing the analyses, it is not necessary to cancel them to satisfactorily meet the updated citizenship law. It could be more efficient to make them fulfill the *acceptable history* constraint. To do so, we define the temporary protocol in Figure 5 as adding the transition *submitBirthCertificate* before the final state. And then, according to the strategy *Notice-Wait-Receive*, we send to the clients the protocol with the time limit and wait for their replies about their system adjustments.

Figure 6 shows an example of migration process using the static and dynamic analysis methods.

1. If protocol P.F can replace protocol P.I, migrate all active instances to P.F.
2. If not, first, with static analysis, classify active instances as follows:
 - (a) Identify states of the sub-protocol that are not affected by changes. Migrate instances at these states to P.F.
 - (b) Identify states of the sub-protocol that might cause the violation of acceptable history constraint. The instances at these states will be further examined in 3.(a).
 - (c) Identify states of the sub-protocol that might cause the violation of correct interaction constraint. The instances at these states will be further examined in 3.(b).
3. second, with dynamic analysis, classify active instances as follows:
 - (a) Using actual histories of instances, extract migrateable instances from the group of instances at the states identified by 2.(b). Migrate them to P.F.
 - (b) Using client protocols, extract migrateable instances from the group of instances at the states identified by 2.(c). Migrate them to P.F..
 - (c) Using instance properties, extract migrateable instances from the instances not filtered by the above methods. Migrate them to P.F.
4. Deal with the non-filtered instances using one of the approaches described in Section 3.3.

Figure 6: Migration process.

4 Architecture and Implementation

To illustrate the viability of our approach presented in this paper, we have developed a prototype for performing the creation and modification of business protocols, analyzing the impact of protocol changes, and migrating ongoing instances to a new protocol. The prototype has been implemented as a part of ServiceMosaic project (servicemosaic.isima.fr), which is a CASE toolset for modelling, analysing, and managing service models including business protocols, orchestrations, and adapters. The prototype architecture (see Figure 7) consists of a *protocol manager* and a *protocol evolution manager (PEM)*. These modules have been implemented using Java in the Eclipse platform (www.eclipse.org).

For the understanding of the Web services protocols management and execution environments related to the protocol evolution, below the dotted line, we briefly show the infrastructure (called *SELF-SERVE* service development platform) for registering, discovering, and composing Web services. A description of SELF-SERVE prototype can be found in (Sheng, Benatallah, Dumas & Mak 2002).

4.1 Protocol Manager

The *protocol builder* assists protocol designers in creating protocol definitions and editing existing ones.

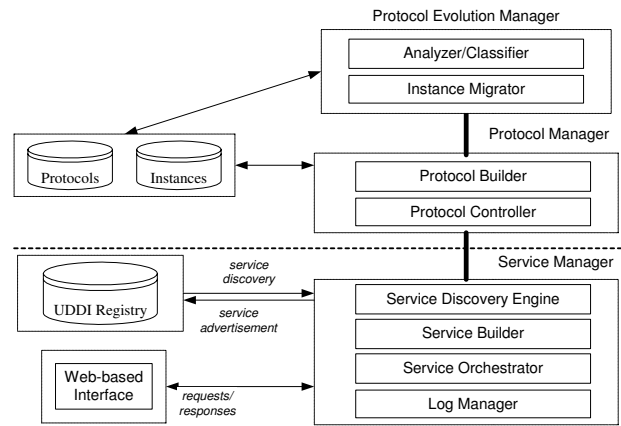


Figure 7: Prototypical architecture

A protocol definition can be edited through a visual interface and exported either to an XML document or to a relational database for subsequent processing. The protocol builder offers an editor for describing a state machine diagram of a protocol. It also provides means to describe the properties of states (e.g., state ID, state name) and transitions (e.g. transition name). The protocol builder uses the XML representations of models to generate control tables which provides the information required by the protocol controller.

The *protocol controller* reads the protocols created by the protocol builder, maintains protocol's state and checks whether messages are received and sent in accordance with protocol definitions. The controller returns error messages to clients when messages are not compliant. It extracts the knowledge required to conduct its tasks from the control tables mentioned above.

4.2 Protocol Evolution Manager (PEM)

The PEM is implemented as a GUI tool for protocol designers to analyze the change impact (by the *analyzer* component) and migrate active instances to a new protocol (by the *migrator* component). For the analyses, the analyzer/classifier loads up the old and new protocols, as well as the current active instances from the DB, and then performs a variety of analyses and presents the results to the protocol designer so that she can examine the results and use this information to take action. After the analyses, it also presents the possible migrate strategies that can be applied to the groups of instances.

A screenshot of the PEM tool is shown in Figure 8. Here, we see the old protocol on top and the new one below. The current active instances are displayed on the left pane, and migrated instances on the right pane. The result of analysis is shown in the bottom part of the window. In this example, the *Unaffected SPA* is done and the states of the sub-protocol are colored as green, i.e., *ApplicationReady*, *BCSubmitted* and *Cancelled*. In addition, the instances in this sub-protocol are highlighted in the left pane. The result of this analysis shows that the protocol designer can choose the strategy *Terminate* or *Migration to a new protocol* as the possible migration strategies.

The tool will help protocol designers or service administrators in managing the protocol evolution problem by simplifying and automating complex evolution-related operations. Using this tool, the users are able to:

- Load and show old and new protocols, and current active instances from DB.

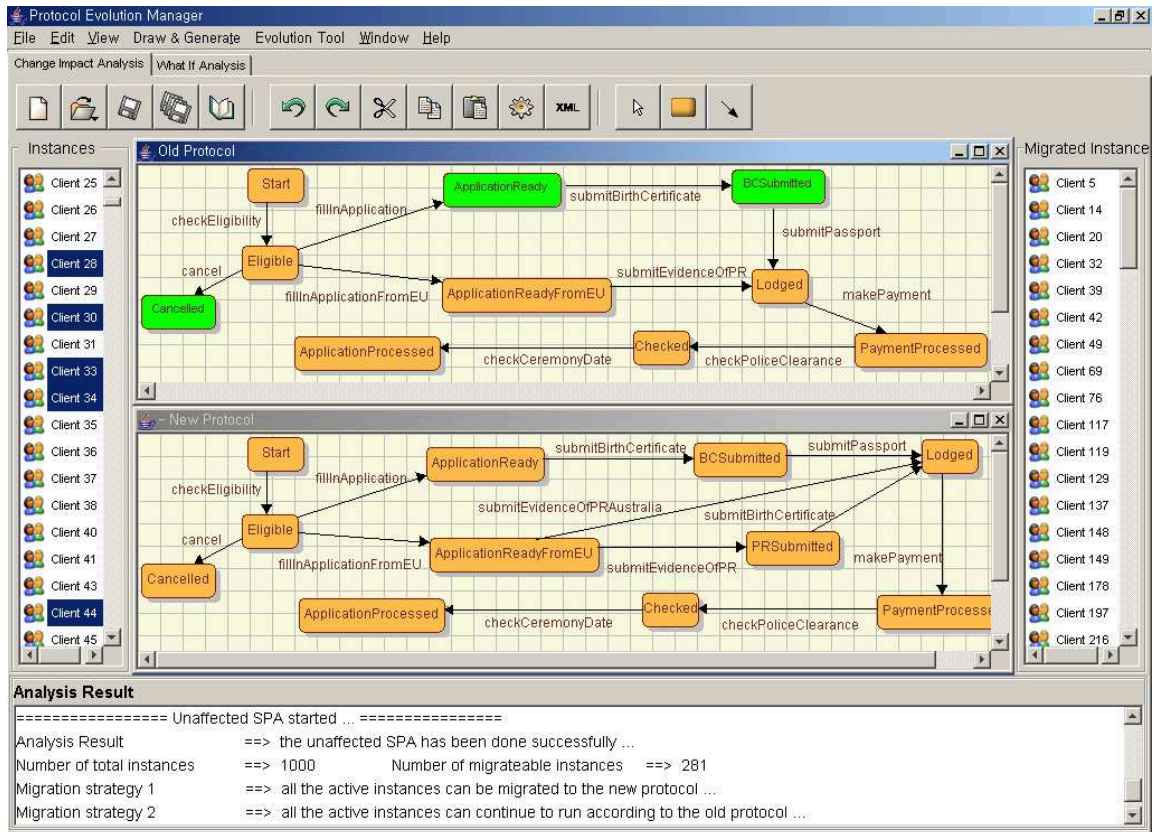


Figure 8: Screenshot of Protocol Evolution Manager

- Choose a particular state and show the instances at the state.
- Choose a certain instance, show its current state, and the history taken by it.
- Show a client’s protocol and the interaction path between the service and client protocol.
- Perform the four types of analysis at the protocol level and show the sub-protocols in different color. Highlight instances belonging to a certain sub-protocol.
- Perform the three types of analysis at the instance level, and show the instances from the analysis result.
- Migrate classified instances to the new protocol, and show several migration statistics, e.g., percentage of migration.

However, in case that the clients’ protocol are not available, this tool cannot perform the future path analysis, since we assumed that we can know them. In addition, this tool does not support the clustering of non-migrateable instances. Hence, protocol designers have to manually look at the instances and handle them individually.

5 Related Work

The protocol evolution is related to three other evolution problems: database schema evolution, software component evolution, and workflow evolution.

Database schema evolution: The database community has considered the problem of managing schema evolution, mainly in the field of object-oriented databases (Andany, Leonard & Palisser 1991, Bertino & Martino 1993, Ferrandina, Meyer & Zicari 1994, Estublier &

Nacer 2000). To meet the new requirements of database applications, the schema definition is changed over time, by adding or removing schema elements. The work in this area has developed several techniques that support the mapping of schema elements from the old to the new schema. Such approaches include the conversion, which transforms the data of the database to make them comply with the modified schema, and the class versioning, which allows the old started applications to continue to use the old schema. However, the business protocol evolution differs from the database schema evolution in two significant ways. First, the database cannot be accessed and ongoing transactions get blocked during the database reorganization, while protocol evolution needs to migrate ongoing instances to the new protocol without stopping the system, blocking and restarting them. Second, in the case of class versioning it is acceptable for old applications to run according to the old schema, whereas there might be situations where it is not possible for ongoing instances to continue to run according to the old protocol, e.g., in case that there are security holes in the definition of security protocol. Therefore, we are unable to use techniques for database schema evolution to this context.

Software component evolution: Software component evolution has been considered important for getting the benefits of component-based software developments, such as component reuse, easy maintenance, and greater flexibility. The components can be evolved since they are hardly flawless. The evolution is a result of satisfying new application requirements, ranging from software structure changes to problems and bug fixes. Most solutions to this problem are based on versioning mechanisms (Levine 1999, Englander 2001, Eisenbach, Jurisic & Sadler 2003). To give the version information, they support enhancing the filenames of the libraries by version numbers

or the libraries by special meta-files (e.g., manifest files of an XML format). So, such mechanisms enable multiple versions of a component to exist in a system and allow applications to use different versions of one component. However, these approaches are not applicable to our problem, for the objective of our work is different from one of them.

Workflow evolution: The protocol evolution has some similarities with workflow evolution (Ellis, Keddara & Rozenberg 1995, Casati, Ceri, Pernici & Pozzi 1998, Sadiq 2000). Ellis et al. coined the issues of dynamic workflow change in their work (Ellis et al. 1995). They exploited a Petri net abstraction for modeling dynamic change which means the change "on the fly" while workflow instances are running. Their approach is based on "a change region" that contains the parts of the Petri net directly affected by the change. However, their approach does not provide a method for calculating the change region, i.e., the change region should be determined manually.

Casati et al. presents a set of basic operations that allow the modification of a workflow schema and preserve structural and behavioral correctness when they are applied to the workflow modification. They also propose evolution policies applicable to ongoing workflow instances. However, the grouping of instances done according to the policies is left up to the workflow designer. Compared to their work, the grouping and classifying instances can be done automatically with the static and dynamic analysis methods.

To handle active workflow instances, Sadiq proposes modification policies which can be adopted by the workflow designer. And then, Sadiq introduces a three-phase modification process consisting of defining, conforming and enacting the modification. In the definition phase, the modification of workflow schema is done. During the second phase, active instances are grouped on the basis of compliance with the new schema. As a result, the compliant instances and non-compliant instances are determined. For the non-compliant instances, Sadiq proposes the compliance graph that acts as a bridge between the old and new schemas. The third phase of the modification process is to enact the modification and migrate instances. However, it is not detailed how to determine the compliance of instances with the new schema. In addition, compared to their two types of grouping methods, our framework allows a protocol designer to perform more fine-grained partitioning and choose more variety of migration strategies.

Web service versioning: In the context of Web services, some recent work (Brown & Ellis 2004, Kaminski, Muller & Litoiu 2006) has proposed the versioning techniques for managing the problem of Web service evolution. Brown et al. proposed an approach based on the use of version namespace and the use of version numbers in UDDI entry. The approach allows multiple versions of a Web service to support client services that are dependent on earlier versions of the service. Kaminski et al. presented a design technique called Chain of Adapters to handle the problem of managing the Web service version and achieve the backward compatibility with clients written to work with older versions of the Web service.

6 Conclusions and Future Work

This paper provided an approach to tackle the problem of static and dynamic protocol evolution. In particular, we identified constraints that can be used as *good* characteristics for the management of dynamic protocol evolution. In addition, to analyze the impact

of protocol changes, we presented the static analysis methods, based on a protocol replaceability and three types of SPAs, and the dynamic analysis methods, based on three additional knowledge (i.e., future path, past path, and instance property). According to the analyses, the grouping (classification) of ongoing instances is automatically performed by our developed tool, rather than manually by the protocol designer. Namely, the main result of this paper is that we have presented a comprehensive approach to the protocol evolution management where we provide a formal model, operators, and tool support for migrating active instances from the old to the new protocol without generating problems such as the violations of the identified constraints.

In future work, the semantic equivalence of protocol changes will be addressed, such as swapping two messages, splitting one message into two messages, and removing messages. In addition, we plan to extend the change impact analysis to what-if analysis and other types of analysis, that help protocol designers, service administrators and business users to plan the protocol changes and improve the quality of their services to their business partners. For example, after protocol changes, how many clients cannot be migrated to the new protocol and, as a result of such changes, how the business profit is affected by the changes, in case that the protocol is relevant to business transactions. Secondly, we plan to identify areas of improvements in business protocol definitions and exploit the knowledge generated by the analyses in the context of services optimization. Finally, providing a variety of these analyses is not straightforward, since there exist many different types of analyses for users to conduct, and it is difficult to satisfy their needs by predefining some queries. Hence, we plan to provide OLAP-style functionalities for services administrators and protocol designers to perform the analyses fit to their needs.

References

- Alonso, G., Casati, F., Kuno, H. & Machiraju, V. (2004), *Web Services - Concepts, Architectures and Application*, Springer-Verlag, 354 pages.
- Benatallah, B., Casati, F. & Toumani, F. (2005), *Representing, Analysing, and Managing Web Service Protocols*, 'Data and Knowledge Engineering Journal (DKE)', Elsevier Science.
- Benatallah, B., Casati, F. & Toumani, F. (2004), *Web Service Conversation Modeling: A Cornerstone for e-Business Automation*, 'IEEE Internet Computing', 6(1).
- Benatallah, B., Casati, F., Grigori, D., Motahari Nezhad, H. & Toumani, F. (2005), *Developing Adapters for Web Services Integration*, 'CAiSE 2005', Springer-Verlag.
- Skogsrud, H., Benatallah, B. & Casati, F. (2004), *A Trust Negotiation System for Digital Library Web Services*, 'International Journal Digital Libraries', vol. 4.
- Sheng, Q.Z., Benatallah, B., Dumas, M. & Mak, E. (2002), *SELF-SERV: A Platform for Rapid Composition of Web Services in a Peer-to-Peer Environment*, *in* 'Proceedings of the 28th Very Large Data Base Conference (VLDB'02)', China.
- Bertino, E. & Martino, F. (1993), *Object-Oriented Database Systems: Concepts and Architecture*, Addison-Wesley.

- Andany, J., Leonard, M. & Palisser, C. (1991), Management Of Schema Evolution In Databases, *in* 'Proceedings of the 17th Very Large Data Base Conference (VLDB'91)', Spain.
- Ferrandina, F, Meyer, T. & Zicari, R. (1994), Implementation lazy database updates for an object database system, *in* 'Proceedings of the 20th Very Large Data Base Conference (VLDB'94)', Chile.
- Estublier, J. & Nacer, M. (2000), Schema Evolution in Software Engineering Databases – A new Approach in Adele environment, 'CAI Computer and Artificial Intelligence Journal', Vol 19. pages 183-203.
- Levine, J. R. (1999), Linkers & Loaders, Morgan Kaufmann, vol. 1.
- Eisenbach, S, Jurisic, V. & Sadler, C. (2003), Managing the evolution in .NET programs, *in* 'Proceedings of the 6th IFIP International Conference on Formal Methods for Open Object-based Distributed Systems', France.
- Englander, R. (2001), Developing Java Beans, O'Reilly.
- Ellis, C., Keddara, K. & Rozenberg, G. (1995), Dynamic Change within Workflow Systems, *in* 'Proceedings. of the Conference on ORganizational Computing Systems', ACM Press.
- Casati, F., Ceri, S., Pernici, B. & Pozzi, G. (1998), Workflow Evolution, 'Data and Knowledge Engineering'.
- Sadiq, S. (2000), Handling Dynamic Schema Change in Process Models, 'Proceedings of the 11th Australian Database Conference'.
- Brown, K. & Ellis, M. (2004), Best practices for Web services versioning, 'IBM Technical Report'.
- Kaminski, P., Muller, H. & Litoiu, M. (2006), A design for adaptive web service evolution, 'In SEAMS'06', pages 86-92.