Diploma thesis

INFERENCE OF UI ANNOTATIONS FROM BPEL PROCESSES AND XML SCHEMA

Tri Vo Hoang

01.01.2010 – 20.06.2010

Dresden University of Technology
Department of Computer Science
Institute for Systems Architecture
Chair of Computer Networks
01062 Dresden – Germany

Chair owner Prof. Dr.rer.nat.habil.Dr.h.c. Alexander Schill
Responsible assistant Dipl.-Inf. Marius Feldmann
I hereby certify that this document has been written by myself, and describes my own work about “Inference of UI annotations from bpel processes and XML Schema”, unless otherwise acknowledged in the text. All verbatim extracts have been distinguished by quotation marks, and all sources of information have been specifically acknowledged by clear cross-referencing to author and work. I declare that this work has not been submitted for any other degree.

Vo, Hoang Tri
Dresden, 20.06.2010
# Table of Contents

1. Introductions .............................................................................................................. 5  
   1.1 Motivation ........................................................................................................ 5  
   1.2 Goals ............................................................................................................... 5  
   1.3 Chapter overview ............................................................................................... 6  
2. Basics ....................................................................................................................... 9  
   2.1 User Interfaces for Service Oriented Architectures ........................................ 9  
      2.1.1 Overview .................................................................................................. 9  
      2.1.2 Traditional Approach ............................................................................ 10  
      2.1.3 Model Driven Approach ....................................................................... 11  
   2.2 BPEL .................................................................................................................. 15  
      2.2.1 Relevant elements overview ................................................................ 15  
      2.2.2 Relevant elements in details .................................................................. 16  
3. Requirements analyses ............................................................................................ 20  
   3.1 Goal description ............................................................................................... 20  
   3.2 Scenario .......................................................................................................... 21  
   3.3 Requirements analyses ................................................................................... 22  
      3.3.1 A chain of data flow problem ................................................................. 22  
      3.3.2 Structured activities problem ................................................................. 25  
      3.3.3 UI annotation conflicts ........................................................................... 26  
   3.4 Summary of requirements ............................................................................... 27  
4. Related work ........................................................................................................... 29  
   4.1 Overview .......................................................................................................... 29  
   4.2 Concurrent Single Static Assignment Form .................................................... 30  
   4.3 Deriving Explicit Data Links in WS-BPEL ....................................................... 31  
5. Solution concepts .................................................................................................... 35  
   5.1 Algorithm Fundamentals ................................................................................. 35  
      5.1.1 Serialization Process ............................................................................. 35  
      5.1.2 Solution concepts overview ................................................................... 35  
   5.2 A node model for matching algorithm ............................................................. 38
1. INTRODUCTIONS

1.1 Motivation

Nowadays software applications running in the internet environment must deal with complex problems such as system scalability, interoperability, reliability, and application integration. Service-oriented architectures (SOA) are an approach to deal with these increasing complexities. Its principle of re-usable, well-defined services and loose coupling ensure a scalable solution to the above problems.

However, a question emerges. Does SOA treat the User Interface (UI) as a reusable resource in the same way as it does with its business logic? Assuming the UI is treated just as a single item, it would be programmed over and over again in every consumer’s service until discarded then re-developed from scratch. It is such a waste.

Some top-rated books like [7] provide comprehensive description of SOA concepts however miss an important question: whether the UI is built into the services or not. Another one [8] meanwhile states that the UI stays out of the service. Indeed, some approaches propose to automatically generate the UI by inferring XML Schema and operations specified in the service interface. However, the UIs generated in this manner inherently suffer from usability issues [1]. Another approach for the UI generation is an annotation-based approach known as the ServFace project [2] developed by SAP, TU Dresden and Lyria. It annotates the WSDL [3] of a service with UI related information and then the information can be exploited for the generation of UIs for interactive service-based applications. Recently it can be used to annotate the semi-automatic UI generation of a specific web service.

The semi-automatic UI generation of a specific web service must be now coupled the second requirement: “UI orchestration”. Different web services (each with their own UI annotations) are now brought to play in altogether to compose in one new service. They pose an interesting challenge: how to automatically generate the new web service’s UI annotations without any involvements from the service providers.

The current progress of this ServFace project depends on an effective authoring tool which eventually enables the generation of UI annotations for a single web service. So far, there are no researches on such automation for the composed web service.

1.2 Goals

Inspired by the requirement of UI annotations for the composition of web services, this paper primarily deals with the following questions:
Key question 1: How to generate annotations for the composed web service?

The motivation aforementioned challenges us to develop concepts to re-use the UI annotations of the partner services for the composed web service; hence the UI of the composed web service does not have to be designed from scratch. This implies to derive explicit data dependencies from different web services in the service composition described by WSBPEL [3].

Key question 2: How to derive the data dependencies in the BPEL?

Deriving the data dependencies in the BPEL requires the “Def-Use” analysis [30], which is the relationship in data-flow analysis, known as the relationship of data writing and reading on a variable. Inside the BPEL, data from external services is written to variables and transferred internally between them until it is used in an operation. This paper focuses on the “Def-Use” analysis with the expectation of complex data flows in the BPEL.

Key question 3: Which limitations of static analysis are expected?

Corresponding to a service operation, a UI layout has to be prepared for the input parameters that the user enters before the operation can be executed. For this reason, the UI annotations of the composed web service have to be generated before the execution of the service in the BPEL engine, so that they are ready for displaying to the client GUI. It requires analyzing the data dependencies between web services “statically”.

Static analysis is the analysis performing without executing programs. Since the data flows in the BPEL engine depends on the instance data in runtime, the analysis encounters several problems in gathering information about the possible set of values calculated at various points for the data dependencies. In short, during the static analysis, there may be several possible writers on the same variable. As a result, there may be more than one UI annotations for a single parameter element of the composed web service. The analysis has to deal with such conflict situations between possible writers on a variable, between possible UI annotations for a parameter element of the composed web service. And it is much more difficult to handle them in the offline environment.

1.3 Chapter overview

The following description will help readers with an overview and guide them how to read the document more effectively.

Chapter 2: Basics (optional read)

The 1st part of the chapter introduces the development of GUI for web services with several traditional and modern-driven approaches. It simply gives readers an
overview picture in this area and finally positions the annotation-based approach, which is related to the work of this paper.

The 2nd part summarizes the BPEL 2.0 specifications [3]. It does not tend to cover all specifications, but focuses only on some elements of the BPEL, which take important roles in the data flow analysis and in the solution concepts of this paper. It is recommended to take a quick view on section 2.2.1 for relevant elements of the BPEL.

Chapter 3: Requirements analyses

This chapter is very important. It describes not only the main goals but also the requirements to archive them in a test scenario. Without understanding the problems, a good solution concept cannot be done successfully and effectively. Three main problems will be stated throughout the work: the chain of data flow, the static analysis of BPEL structured activities and annotation conflicts handler.

Chapter 4: Related work (optional read)

After understanding the requirements, this chapter will compare the work in this paper with other similar works. Recently there are no approaches which focus on the same target as this work, except for some near approaches in the area of compiler construction and optimization for the BPEL. But these similar works are interesting due to their static analysis of the data flows, of data writing and reading on the BPEL’s variables. Exploring and comparing them with the solution concept will help understanding the difficulties in the data flow analysis and to optimize it better.

Chapter 5: Concepts

The whole solution concepts might be long to read and hard to keep track of. The algorithm fundamentals in section 5.1 are important. It covers all concepts at the first glance. Concepts will be divided in 4 main parts: a node model, a matching algorithm, a copy process with conflicts handler, and a BPEL modifier.
Chapter 6: Implementation

Implementation of concepts in details

Chapter 7: Evaluation

The chapter begins by revisiting the test scenario in the requirements analysis. The test will explain 4 parts of the concepts with concrete examples. Finally it is followed by a discussion on the advantages and disadvantages of the concept in section 7.2.
2. BASICS

2.1 User Interfaces for Service Oriented Architectures

2.1.1 Overview

A service-oriented architecture (SOA) can provide enterprises with significant benefits. One of the strong benefits is the ability to reuse application functionality and to interconnect heterogeneous applications to create new composite applications [10]. By the current standards of web services, SOA has successfully presented an application with a simple interface that can abstract away its underlying complexity. Hence, users can easily access the services without knowledge of the platform implementation and conveniently integrate them with other resources. The partners and the service providers, altogether, can link up a chain of business services.

The standards of web services have provided an abstract interface of the application logic, but without the graphic user interface (GUI), thus every partner in the chain has to implement it manually. The development of GUIs is one of the most time and capital consuming aspects in software development. This significant hurdle needs to be cleared in order to gain the full benefits of an SOA. Moreover, the users also need an interface through which they can efficiently interact with the resources.

For this purpose, a number of researches in this area appear to target the SOA user interface space. In the following sections, several approaches for the development of GUI and its flexible combination together with functional services will be shortly discussed. They range from a traditional approach, which is the manual implementation of the GUI in a specific platform, to a model-driven approach, which is an automatic generation of codes from a model to several target platforms.

In the traditional approach, an overview over the desktop client, web-based and rich portal interfaces is introduced. It will help understanding the trend in the UI’s development before going to the model-driven approach.

In the model-driven approach, there is a need to examine in the mechanism of the XML Schema inference [17], which generates the GUI from the Schema description automatically. But this mechanism also has some limitations in the generation of GUI and is later enhanced by further developments, such as an inference mechanism in compound with the semantic description [18].

However, the GUIs generated in this way are not sufficient and do suffer from usability issues [1]. To facilitate the automation, an abstract UI-model can be defined and then transformed into a concrete model for execution in a specific platform [24] [25]. It simply helps the readers to finally position the annotation-based approach (ServFace) [1] [2] in the whole picture.
2.1.2 Traditional Approach

In SOA, a service is described using Web Service Description Language (WSDL) [4]. Based on this WSDL a stub is generated for the client automatically, and then the client can invoke the remote service by calling the method in the generated stub [7]. In the traditional approach, although the stub is generated automatically, the GUI is designed to obtain input information and present the service output to the users with a certain programming language and in a specific platform manually.

![Figure 1: Traditional approach for generating GUI](image)

Development of GUI for web services on various platforms has different problems:

**A desktop UI** offers the rich and most interactive way for the end users, but it takes a considerable amount of time to develop, deploy and maintain. Furthermore the desktop applications may not be compatible on all platforms and they are not available in locations other than their desktops.

**A web-based UI** solves the problems (as above) with the desktop applications. It requires no client deployment, and the user access is available from anywhere. But its interactivity is limited by the constant refreshing and contacting with web servers, which significantly slows down its responsiveness [11]. But Asynchronous JavaScript and XML (Ajax) has led to Rich Internet Applications (RIAs), providing a rich UI experience to applications delivered via the Web browser. It extends the traditional browsers to more complex solutions by enabling asynchronous communications and reducing the whole page refreshes as a result.

**Portals** can render and aggregate information from different sources and display them through a unified interface. However, the traditional “data-oriented” web services require aggregating applications to provide specific presentation logic for each of these web services. The Web Services for Remote Portlets (WSRP) [12] standard solves this problem by introducing a set of “presentation-oriented” [13] web service interfaces, which provides a common protocol for both application logic and presentation logic so that the web services can be plugged and played with the portals.
The figure below (on the left) illustrates a typical data-oriented web service that provides unformatted data. It must rely entirely on the client-side for executing the view. On the right side is a WSRP service with its distributed presentation logic, delivering UI components through Web services.

![Diagram of Data-oriented Web service and Presentation-oriented Web service](image)

**Figure 2: Data oriented (left) and presentation-oriented web service (right) [13]**

**Advantages/Disadvantages:** in general, by delivering UI components through Web services, portals can aggregate portlets into a unified UI dynamically without UI programming. It reduces the costs for manual UI implementation on the client-side and thus improves time-to-value for a software developer. However, WSRP portlets cannot use external style sheet provided by a WSRP consumer. Although the WSRP specifies a standard set of CSS styles that can be defined by WSRP consumers, the usability issues remain uncertain. In generic WSRP portlets, you have less control over the UI [14].

### 2.1.3 Model Driven Approach

Going against the advantages of RIA and Portals, a new problem occurs when the ever-changing enterprising businesses of today demand a quick adaptation of their applications' UIs to fit these changes, specifically, to support a new platform or a new language. For this reason, the Model Driven Approach is taken into account.

Model Driven, in general, separates the model from codes. The developer works on a platform independent model, selects the specific target platform, and the tool generates the codes or the codes are interpreted in a runtime environment [15]. In the aspect of automatic UI generation, it is the question to generate GUI from data schemas automatically, or to define a model description of GUI, which can be generated to multi target UI in runtime. They will be mentioned shortly in the following sections.
2.1.3.1 XML Schema Inference

XML Schema describes data structures, types and constraints. They show several correspondences to the form of GUI. XML Schema Inference is meant to transfer information derived from XML Schema domain into visual representation [17]. The works of [16] and [17] show many mechanisms which will generate XForms [20] according to the schema.

1. The WSDL document tag `wsdl:documentation` can be used as label or hint text for describing an operation, message and service respectively. The structure of a XML element `complexType` and its children elements can be grouped together in the UI.

2. XML Schema has a lot of built-in data types. In general, any simple type can be displayed as an `input` field (8.1.2) [20]. However users may type in a wrong format and the type safety will break. To help the users with a more advanced GUI, the input layout according to the built-in data type can be generated. For example: `xsd:boolean` can be displayed as checkboxes.

3. XSD Restrictions can give additional information about the limitation of an XML element to a set of given values. For example, `xsd:enumeration` can be displayed as a selected drop-down box.

Disadvantages: XML Schema is an important concept, it is proposed to be provided as the base for an automatic GUI generation mechanisms. But it heavily depends on the quality of the Schema description. The quality is poor when the Schema uses a weak type format (`xsd:string` for a date input), or missing semantic (`xsd:string` for password input) and when there are no value restrictions. The WSDL documentation tag is also an optional element and is rarely used. Moreover, the automatic generation of WSDL based on the bottom-up/code-first design makes its quality only worse.

Additional specifications, such as the Web Ontology Language (OWL) [21], can be provided to guide a better GUI generation [18]. An example is the generalization, otherwise known as the relationship between sub class and super class. Resources that share the same super class can be grouped together in the GUI. In general, semantic specifications are useful in building a hierarchical structure for the GUI, but they are not enough for creating executable interactive applications.

2.1.3.2 Abstract UI-models approach

The limitations of such works like WSGUI in [22] [23] usually involve the direct mapping mechanisms between the web service description interface WSDL and the implementation language (XForms) in a specific platform (a web browser). However, their limitations do not pertain to devices with different languages and multi platforms.
**Advantages:** if the GUI is considered at an abstract level, it brings across many benefits: firstly, the designers do not have to learn the intricate details of the widely possible implementation languages supported by the various devices. Secondly, the GUI is not always tied to a particular platform and hence automatically provides a transformation between platform-specific models.

For this particular goal, many pieces of works in both Teresa [24] and UsiXML [25] have developed model-based user interface descriptions in XML based language, through which the abstract user interface (AUI) and concrete user interface (CUI) can be defined. By exploiting these descriptions, associated transformations for the target devices are generated. While Teresa develops transformation mechanism to map one model onto another, the logics and the definition of transformation rules are completely hard coded with little or no control by designers [25]. The UsiXML description language and its authoring tool GrafiXML [26] enable a flexible expression and manipulation of the model at each stage of development for the designer.

**Disadvantages:** the most notable benefit of model-based user interface description is the automatic transformation between models. But this model-driven approach tightly integrates the relation between AUI and CUI models in its MDD-chain. Furthermore, this approach is very demanding of the designers’ skills and experience in not only the model transformation but also in one particular model description language (UsiXML for example).

### 2.1.3.3 ServFace, an annotation-based approach

As explained in the previous section, the abstract UI-models approach faces the disadvantages, primarily the tight integration between models and the demanding skill requirement for developers. Surprisingly, capable of turning these disadvantages into its own advantages, the annotation-based approach presents yet another aspect of the model-driven in the UI generation to be considered. It focuses on the presentation front-ends, rather than application logic, to “annotate” each element of the web service’s operation using the interactive UI related information. And thus the approach presents a loose coupling of model-driven techniques and Service annotations [27] as described shortly as follow.

a. **Annotation model [2]**

Every parameter element of the service operation is annotated with UI related information. Depending on this annotation information, the GUI for this web service operation can be generated automatically. There are 22 annotations which altogether provide UIs with dynamic and interactive features from the suggestion functionality, a client-side validation of input parameters to an automatic form completion for frequently used text fields.
The structure of annotation model is organized as follow. The root element is the AnnotationModel class that contains a set of ServiceModelElements, which hold the references to specific web service element in a service description. Each ServiceModelElement can contain a set of Annotations, which are the UI related information for this web service element. Each annotation elements can reference to a specific platform or language.

a. Service Composition at the Presentation Layer

When each service already has its UI related information annotated, adopting the idea of integration at the presentation layer for mashups applications (web feeds, web pages), different web services with their UI related information can be easily aggregated to provide a brand new composed service. Meanwhile, the UI annotation of the new service is also generated. This annotation process is supported by an authoring tool, the ServFace Builder [28], which is made so user-friendly that even non-programmers can use it to build services composition on their own.

In this section an overview of the topic GUI in SOA is introduced. There are several approaches mentioned together with a discussion on their disadvantages. Model-driven approach provides the benefit of an automatic UI generation for different platforms and is independent from a specific programming language. The inference mechanisms of XML Schema support the automation, but they often have missing semantic information for the UI. The annotation-based approach, however, not only fills up this UI’s missing information in a single web service, but also facilitates the UI generation for the composed web service. Unfortunately, an automation process has not been supported so far.
### 2.2 BPEL

The Business Process Execution Language (“WS-BPEL” or “BPEL”) [3] is an XML-based programming language used to orchestrate behavior between Web services to form a business process. This section won’t cover all specifications of the BPEL 2.0. Instead, it only focuses on some elements of the BPEL, which play important roles in the solution concepts.

#### 2.2.1. Relevant elements overview

![Figure 4: Relevant information from the BPEL](image)

In this paper, the concept used to analyze the data dependencies between parameter elements from a client and its partner web services is independent from the BPEL’s structured activities. In details, it focuses only on how the data is retrieved and delivered at input and output operation, how it is copied and transferred between variables and where it comes from. Apart from that, the concept can also judge if a data flow is under a certain condition behavior.

1. **Input and output variables in the data flow**

   They are BPEL’s variables, which can be defined from `element` or `messageType`, which reference to XSD element or message in the WSDL respectively.

2. **Input and output operations**

   The BPEL only uses incoming operations (receive, onEvent, onMessage) and output going operation (reply) to receive from and reply to a web service respectively. It uses `invoke` to call a web service’s operation in one-way request, or to send and receive message synchronously in two-ways request.

3. **Where do the operations come from?**

   The BPEL uses `partnerLinks` to provide information about the corresponding `portType` of a web service that the operation belongs to.

4. **Internal data transformation**

   The BPEL uses two elements `assign`, `copy` for the data assignment. They have various forms to select a message, part of a message or an element inside the binding variable to copy to another type compatible one.
(5) Conditions

The BPEL can specify several branches for the BPEL process to process in runtime. Each branch is specified by a BPEL condition element, such as if, elseif and else.

2.2.2 Relevant elements in details

The following part will summarize the BPEL specification for these relevant elements with some important notes for the implementation.

- **partnerLinks** (6.0) [3]

A partnerLink is a link from a BPEL process to an external Web service’s portType. Firstly, a partnerLinkType should be defined in the partner WSDL with its role attribute referenced to a specific portType in the WSDL.

```xml
//partnerLinkType in WSDL
<plnk:partnerLinkType name="NCName">
  <plnk:role name="NCName" portType="QName" />
</plnk:partnerLinkType>
```

Then in the BPEL, a partnerLink references to a specific portType via its partnerLinkType and role attributes.

```xml
<partnerLinks>
  <partnerLink name="NCName" partnerLinkType="QName" myRole="NCName"? partnerRole="NCName"?/>
</partnerLinks>
```

If a particular role is mentioned as a myRole, the operations defined in the corresponding portType will be used to call the BPEL process from the client. If it is a partnerRole, the operations will be used to invoke the partner services.

The namespace of partnerLinkType is indicated by the targetNamespace attribute of the corresponding WSDL document (6.1) [3]. It will be used to decide to which WSDL a partnerLink or a variable belongs.

- **variable** (8.1) [3]

In a BPEL process, variables are used to hold incoming messages from a partner, outgoing ones to a partner, or the data required to hold the state of a process instance. BPEL employs three kinds of variable declarations: WSDL message type, XML Schema type (simple or complex), and XML Schema element.

```xml
//variable can be messageType, XSD type or element.
<variable name="BPELVariableName" messageType="QName"? type="QName"? element="QName"?"+
  from-spec/>
</variable>
```
(10.3-10.4) [3], a BPEL’s variable can be a WSDL message type, XSD type, or XSD element. However, a BPEL’s variable, when used by an operation of a web service such as receive, reply and invoke, must be a WSDL message type. Only when the message type has one single part defined from XSD element, then the variable can be defined from this XSD element.

- **receive** (10.4) [3]

The BPEL process receives input message from a client or from its partner’s WS. Input message is defined by specifying the **partnerLink** and commanding the **operation** attribute to wait for the correct input message. The message is then assigned to the BPEL’s variable which declared by the **variable** attribute. Note that the **portType** is only optional attribute.

```xml
//input message from a specific operation’s web service is assigned to BPEL’s variable.
<receive partnerLink="NCName" portType="QName"?
   operation="NCName" variable="BPELVariableName" …>
   <fromParts>?
      <fromPart part="NCName" toVariable="BPELVariableName" />+
   </fromParts>
</receive>
```

If the **receive** activity is specified with **frompart**, only this part in the incoming message is assigned to the BPEL’s variable (**toVariable**), not the whole incoming message.

**onMessage** (11.5) [3] and **onEvent** (12.7) [3] are similar to a receive activity, in that it waits for the receipt of an inbound message.

- **reply** (10.4) [3]

It is the outgoing message from BPEL process to a client. It has the same syntax as the **receive** activity, except for the optional **toPart**. The usage of **toPart** is totally different from that of **fromPart** and sometimes can be confusing. **toPart** specifies a virtual assignment, which copies the BPEL’s variable in **fromVariable** onto a specific part of the message in the operation, not to the whole message of the operation. When **toPart** is used, it requires the BPEL to specify all assignments for all parts existing in the message of the operation, not just only one part of it.

```xml
<reply …>
   <toParts>?
      <toPart part="NCName" fromVariable="BPELVariableName" />+
   </toParts>
</reply>
```
• **invoke** (10.3) [3]

Invoke operation is used to call on a partner’s operation with the given inputVariable and save the return message from the partner in an outputVariable. Sometimes a BPEL designer can use invoke to send a one-way “response” by invoking the corresponding one-way operation of the service.

```xml
<invoke partnerLink="NCName"
   portType="QName"? operation="NCName"
   inputVariable="BPELVariableName"?
   outputVariable="BPELVariableName"?>
   ...
   <toParts>?
     <toPart part="NCName" fromVariable="BPELVariableName" />+
   </toParts>
   <fromParts>?
     <fromPart part="NCName" toVariable="BPELVariableName" />+
   </fromParts>
</invoke>
```

In the syntax above, fromPart and toPart have the same meaning as the one declared in the receive/reply activity above.

• **assign** (8.4) [3]

An assign activity holds several assignments copy. Each of them performs a copy from a type-compatible value from the source from-spec to the destination to-spec. Note that keepSrcElementName attribute specifies, whether the element name of the destination to-spec will be replaced by the element name of the source from-spec during the copy operation.

```xml
<assign>
   (<copy>
      <from keepSrcElementName="yes|no">...</from>
      <to>...</to>
   </copy>)+
</assign>
```

(8.4.3) [3] for a copy operation to be valid, the copy “from” and copy “to” selection must be in type compatible. The BPEL standard dictates that it is possible to assign an XSD element to an XSD element (of a compatible element), or an XSD type to an XSD type (of a compatible type) or a WSDL message to a WSDL message (of a compatible message). Two WSDL message types are the compatible if their QNames are equal, by definition.

It is important to know that a copy operation has various forms. The following syntax illustrates there are various assignment forms to resolve during implementation:
The 1st form specifies the assignment of variable’s property (7.3) [3]. A variable’s property is defined by an element, a type or part of a message in the WSDL.

The 2nd form is used to initiate a new variable in the BPEL. The literal value does not have any UI-related annotation information. This assignment is irrelevant.

The 3rd assignment, copies the whole partnerLink to partnerLink.

The 4th form can be used to copy one BPEL’s variable to another. If the variable is a message type, the option part attribute is used to specify a concrete part in the message for the copy. The queryContent is an XPath1.0 expression which selects an element (simple/complex element) inside the binding variable.

The 5th has the same usage as the 4th. It presents the data selection in a text content, which separates variable and part with a dot.

Under 8.4 [3]
//1. Copy variable’s property
<from variable="BPELVariableName" property="QName" />

//2. Copy a literal value to a variable → not relevant, ignore
<from><literal>literal value</literal></from>

//3. Copy the whole partnerLink
<from partnerLink="NCName" endpointReference="myRole|partnerRole"/>

//4. Copy message, message’s part or element.
<from variable="BPELVariableName" part="NCName"?>
  <query queryLanguage="anyURI">?
    queryContent
  </query>
</from>

//5. Copy message, message’s part or element in a query expression.
<from expressionLanguage="anyURI">expression</from>

What are the most important things in the BPEL to remember?

• First of all, the namespace of partnerLinkType is indicated by the value of the targetNamespace attribute of the corresponding WSDL document. During the implementation, this information is useful for deciding, to which WSDL an operation or a variable belongs. Knowing where they come from is also useful for the task of copying UI annotation.

• BPEL’s variable can be defined from a WSDL message type, XSD type or an element. But the BPEL process can copy the whole message or a single part of the message, or a single XML information item by using XPath query. They are the data flow inside a BPEL process, not just the BPEL’s variable only.

• There are too various forms of assignment to deal with. And selections for the copy operation have to be of compatible types.
3. REQUIREMENTS ANALYSES

3.1 Goal description

This section will describe our objectives and analyze the demands of the whole work. The following annotation of the services with UI related information is developed by the ServFace project, which was acknowledged earlier in chapter 2. Together with the annotation by ServFace project, BPEL is chosen as a description language to describe the composition of different web services. Readers may refer to section 2.2 for details information about BPEL.

To begin we first ignore to talk about the UI annotation in details. Just imagine that there is UI related information, which binds to a concrete web service and describes the out-looking presentation of this web service. There are several web services that you may want to compose into one new web service and each of them has their own UI annotations. The question is how the new web service’s UI annotation can be generated automatically with no involvement from the service providers. Let’s call these binding web services partner’s WS and the composed web service BPEL’s WS. The following figure illustrates the first look at our main goal with one partner’s WS and one BPEL’s WS.

![Figure 5: Copy UI annotations for equivalent parameter elements.](image)

In the figure above, one partner’s web service (on the right) has an UI annotation description for his parameter elements inside an operation; in turn the composed web service (on the left) wants to have the UI annotation description for its web service’s operation too. So if it is known which elements of the two are semantically equivalent with each other, the UI annotation of the partner’s WS can be copied for the BPEL’s WS. The following figure illustrates the situation when two or more partner’s WS come into play together.
With two or more partner’s WS on the right with their UI annotations, they are composed into one new web service on the left. The composition process is described by the BPEL. This composition description language will determine the data flow between the new composed web service and his partners.

3.2 Scenario
To illustrate the idea behind our concepts and to analyze the requirements easier, the following scenario of a travel booking service is introduced:
The “Travel booking service” (TravelBS) is the composed web service for 3 different web services: a “Flight booking service” (FlightBS) and two “Auto booking services” (AutoBS). TravelBS receives a booking request from a client (op. travelBooking), and then it contacts the FlightBS to book a flight (op. bookFlight). If the client needs an auto, the TravelBS asks two AutoBS for the auto price service (op. priceQuery), and books an auto by the service provider with the cheaper price (op. autoBooking). Here we have an additional condition: if the auto service provider does not have any available autos, the whole TravelBS will be compensated with the previous successful FlightBS. Successful or compensated result will both be returned to the client.

In our scenario, the web services FlightBS and AutoBS already have their elements annotated with the UI related information. The scenario is to copy their UI annotations for the TravelBS, based on the BPEL composition description. WSDL are ready for every web services and we have the two AutoBS using the same WSDL, only with different service name and targetNamespace.

3.3 Requirements analyses
3.3.1 A chain of data flow problem

Figure 8: Data received from the client is given to WS1 directly (path 2), or it is joined with another data before it is given to WS2. (path 1.1-1.2-1.3).

1. Description

To find out the relation of elements in the composition of web services, it is necessary to analyze which input elements of which BPEL WS’s operation is used by the partner’s operation (and which output elements of the partner’s operation arrives at the output of the BPEL WS’s operation). But it is not as easy as “his input is the same as mine and so on”. The reason is that, inside a BPEL’s process, the BPEL’s input variable is not only used directly by its partner operation (figure 8, path 2). The input variable can also be copied to another variable completely or
only partially. Even variables from different web services are joined together into a new one, before they are called by an operation of the partner web service (path 1). Moreover the BPEL has several different ways and syntaxes for the copying data, as illustrated below.

![BPEL process as Web Services diagram](image)

**Figure 9:** Four variations for one BPEL assignment.

In every BPEL assignment, it is possible to copy a whole message of an operation to another message, or just one part of it to another, or only one under element of a part of a message to another. The assignment results in different kinds sticking together and building on a chain of data flow, as illustrated by the figure 10 below:

![Chain of data flow](image)

**Figure 10:** chain of data flow makes it more difficult to find out the semantic equivalent.

### 2. Problems

The chain of data flow, exhibited in complex patterns, makes it more difficult for us to locate the semantic equivalents between elements. It may land in a situation like this: following the first assignment, which copies an element X to one element Y, the second one may have a “super set” element (or a “subset” element) of Y copied to another one and so on. This could go on to infinite complication; however, it is only one of the possible cases. But the question is how it is able to determine the actual and uncomplicated relation between “front-end” elements of the web services when the relation itself is subjected to a chain of data flow. The following test scenario will make it clearer.
3. Test scenario

The following situation illustrates the complexity of the chain of data flow, whereby the pair of two relative assignments is not a full transitive conclusion (when a part of the variable is copied to another one, but not the whole of it)

a. 2nd assignment is about a super set element of the 1st assignment

![Figure 11: element “destinationAirport” of the receive variable is semantic equivalent with a child element “place” of the variable “bookAutoInput2”.

In the first assignment, an element destinationAirport of the receive variable TravelBookingInput (element type) is copied to a child element place of a part parameters inside variable bookAutoInput1 (message type). In the next assignment, a part of the message-typed variable bookAutoInput1 is totally copied to variable bookAutoInput2 (element type). Both variable bookAutoInput1 and bookAutoInput2 are used for booking an auto of AutoBS1 and AutoBS2 respectively.

A new concept has to come with a new relation between the element destinationAirport of the receive variable and the element place of the variable bookAutoInput2 of the partner service.

b. 2nd assignment is about a sub set element of the 1st assignment

![Figure 12: element “startAirport” is semantically equivalent with element “start” inside message’s part “bookFlightRequest” of message “bookFlightInput”.

Travel-BookingInput bookFlight-InputTemp

bookAutoInput1 bookAutoInput2

place

place

1

2

destinationAirport

startAirport

startAirport

start (bookFlightInput

.bookFlightRequest)
The following scenario is created to simulate the situation in which the 2nd assignment is about a sub set element of the first one. In details, in the 1st assignment the receive variable TravelBookingInput (element type) is completely copied to the variable bookFlightInputTemp (element type). In the 2nd assignment, only one element startAirport of the variable bookFlightInputTemp is copied to the element start inside a part of the message bookFlightRequest of variable bookFlightInput (message type). The variable bookFlightInput is then used to invoke the operation for booking a flight.

A new concept has to come with a new relation between the element startAirport of the receive variable and the element start of the partner service.

3.3.2 Structured activities problem

![Structured activities diagram]

**1. Description**

The next problems we shall encounter are due to the various structured activities of BPEL: The BPEL can specify some activities which can be executed in sequence, or in parallel. Or activities are repeated while a predicate holds on (while/repeatUntil) or is controlled by a specified counter variable (for each). The whole process can also be blocked and waits pending for a suitable message to arrive and then perform associated activities (pick). It is possible to select exactly one branch of activities from a set of choices to perform (if-else). Finally some activities with their own variables and failure handles are grouped together in a scope (scope).

**2. Problems**

The problem is that the structured activities depend on runtime data. Since the runtime data of the BPEL engine is unknown in static analyses, it is impossible to follow each activity of the data flow as step by step as in the real time process.

For example, it is statically irresolvable if an event (fault, termination, onAlarm or onMessage event) occurs at a certain activity. By occurring at any
time, an event will change the order of the activities in the BPEL process. For all conditional behaviors (if container activities), it is impossible to statically determine which of the contained activities will be executed, if there is more than one of such activities. The conditional behavior depends on XPath expression and it is shown in [29] that it is undecidable whether a given XPath expression is satisfiable or not.

While there are approaches in chapter 4, which analyze the data flow in the BPEL, dependent on the structured activities; the solution concept in this paper will analyze the data flow independent from them.

3.3.3 UI annotation conflicts
Conflicts occur when two or more parameter elements from partner web services are semantically equivalent to at least one element of the composed web service. Annotation conflicts block off the task of locating the right UI annotation for the composed web service.

Figure 14: UI annotation conflicts

1. Conflicts due to duplicated activities of the same web service

Description: when the solution concept (in finding semantically equivalent element) (in chapter 5) ignores the structure of contained activities, it faces a situation that the same operation of a web service is used in different positions of the BPEL process. Duplicate operation with the same input/output variables can cause the algorithm outputs unnecessary duplicate matching records.

Test scenario: in the test scenario, the reply operation travelBooking of the TravelBS appears twice in the BPEL document. One time is to reply with a successful result to the client, another time to reply with a result in a compensation handler. Both reply operations use the same variable TravelBookingOutput as an output.

2. Conflict elements between different operations of one web service

Description: parameter elements of one web service, yet coming from different operations of this web service, are all semantically equivalent with only one parameter element of one operation of the composed web service. This situation
occurs, for instance, when the receive variable from one operation of the client is used for invoking different operations of one partner web service.

**Test scenario:** request information received from the client (op. travelBooking) for the operation priceQuery is used again for the operation autoBooking. The operations priceQuery and autoBooking are both from the same service AutoBS.

UI annotations of parameter elements from different operations (priceQuery and autoBooking) yet of the same web service (AutoBS) are copied for the client’s parameter elements. Conflicts occur from within one partner service.

### 3. Conflicts between different web services

**Description:** parameter elements from different web services are semantically equivalent with one parameter element of the composed web service. It occurs, for example, when the received variable is used to invoke different partners’ operations, or different partners concurrently output to the BPEL’s variable which replies to the client.

**Test scenario:**

1. Between obligatory and optional services

Request information received from the client for booking a flight is used again for booking an auto. UI annotations from different partners (FlightBS and AutoBS) are concurrently copied for the client’s parameter elements. While FlightBS is an obligatory service, AutoBS is only an optional service when the customer wants to book an auto. This type of conflicts occurs between an obligatory and an optional service.

2. Between two optional services

AutoBS1 and AutoBS2 concurrently provide their services to the client. The service providing a cheaper auto will be chosen. Please note that it is statically unknown, which partner service will be invoked in runtime. UI annotations from different partners (AutoBS1 and AutoBS2) are concurrently used on the client GUI. But only those coming from the invoked partner should be chosen to display to the GUI.

### 3.4 Summary of requirements

In summary, to find out the semantic equivalents of parameter elements between the composed web services and those of its partners, a solution concept should satisfy the following requirements:

1. Solvable for every form of the BPEL assignment.
2. Transparent for any BPEL’s structured activities (sequential, parallel, synchronal or repetitive execution process…), because firstly, the runtime
value of a variable instance during the runtime of the BPEL engine is unknown and secondly, it is undecidable whether a given XPath expression is satisfiable or not. Assume that an algorithm is developed which can cover the handling of all the cases of the structured activities in the BPEL, the complexity of that algorithm is massive.

3. Copy UI annotation from partner services with conflicts handler.

To archive such a solution, an algorithm $y=f(x)$ will be developed which has:

1. A “common presentation” $x$, applied for all kinds of BPEL assignment. Weather the assignment is about a message, a part of it or an under element of a part of the message, the algorithm must have a common presentation model of data to operate. The output of the algorithm is the matching lists, which contain all parameters elements derived from a concrete operation of the composed WS and its partners’ WS. Here the parameters elements are semantically equivalent with each other.

2. The Algorithm must be independent from the BPEL’s structured activities. On one hand, an independent algorithm implies that the sequence order of the process can be ignored even if the current runtime process is sequential, parallel or turned over to a compensation process. On the other hand, it satisfies the main goal somehow.

This chapter presents an overview about the objectives and requirements for the solution concept. The objective is to find out the semantic equivalents between parameter elements of web services in the composition process described by the BPEL.

Due to the complexity of its requirements, a matching algorithm is developed (in chapter 5) to consider all kinds of data flows and implicate the final relation between the BPEL’s WS elements and his partners’ WS elements, Meanwhile, this algorithm must be independent from the BPEL’s structured activities. Then the UI annotations of parameter elements from the partner services are copied to the composed service, under the consideration that UI annotation’s conflicts could occur and hinder the process.
4. RELATED WORK

4.1 Overview

The work of ServFace (described in 2.1.3.3) is partially based on former approaches, i.e., the GUIDD annotations [22]. They annotate parameter elements of a web service with UI related information. Both approaches focus on the creation of UI annotations for single services with the help of an authoring tool. There are no automatic processes to generate UI annotations for the composed web service so far. The solution concept in this paper is the first approach ever (chapter 5).

The chapter of requirements analysis concludes that it is necessary to derive the data dependencies between web services from the BPEL specification. In details it is necessary to determine which parameter elements received from the client are used as inputs for which elements of the partner’s services corresponding, and in return, which parameter elements received from the partner’s services are used as outputs for the client. Recently there have been no approaches which seem to strive for on the same objectives as our work, except for some remotely-related approaches in the area of compiler construction and optimization for the BPEL.

Data-flow analysis

In the analysis of computer software, static code analysis is the analysis performing without executing programs. In this area, data-flow analysis is a technique for gathering information about the possible set of values calculated at various points in a computer program [30]. Recently there are approaches performing this technique on the BPEL. Although they have different objectives from our work, the “Def-Use” [30] relationship in data-flow analysis, known as the relationship of data writing and reading on a BPEL’s variable, has obtained a high degree of interests on our part. In particular, it is shown that the following techniques of data-flow analysis depend heavily on the structured activities of the BPEL, which is different from our solution concept as we strive not to do so. Interestingly, exploring these differences will help understanding the problems surfaced in the requirements of structured activities (section 3.3.2), the disadvantages of the new concepts (if there are any) and how to optimize them.

In the following sections, two approaches will be discussed. The first one can find out the data dependencies between variables, but limits the scope by not specifying the matching elements between web services. The second one satisfies this matching requirement partially, but it’s an over approximation due to the limitation of static analysis.
4.2 Concurrent Single Static Assignment Form

Description

The Friedrich-Schiller-University in Jena - Germany and IBM research on [33] developed an algorithm to map every BPEL’s structured activity into a CSSA form for the analysis of non-initial variables in the BPEL depending on initial input variables. The CSSA is described as follows.

“Single Static Assignment Form” (SSA) [31] is used as an intermediate representation in compiler design, in which every variable is assigned exactly once. They are split into versions and behaved like constants (for example, \( v = v - 2 \) will be represented as \( v_2 = v_1 - 2 \)). Every time the program enters a loop (while) or at the end of a conditional behavior (if-else), the out-going variable may have multiple values, which depends on the control flow path that the executing program taken at runtime. To overcome this problem, the SSA presents such variables by a “Φ function”, for example \( v_3 = \Phi (v_1, v_2) \), with its parameters \( v_1 \) and \( v_2 \) are possible values that the variable may take at runtime.

For programs containing concurrency, such as the flow activity in WS-BPEL whereby different processes write on the same variable concurrently, the CSSA [32] extends the SSA with a new \( \pi \)-function (in analogy to the Φ-function mentioned above) to represent all concurrent values of the given variable in parallel activities.

With the background concepts of CSSA, the work on [33] developed an algorithm to map every BPEL’s structured activity into a CSSA form. And then the analysis process performs iterative runs until it reaches the so-called “fix point”, in which the in-states (and the out-states in consequence) do not change, then non-initial variables are all presented dependent from initial variables.

Example

\[
\begin{align*}
\text{true} & \quad x_1 \leftarrow \text{auto1} \\
\text{false} & \quad x_2 \leftarrow \text{auto2} \\
\end{align*}
\]

\[
x_3 \leftarrow \Phi (x_1, x_2)
\]

Figure 15: Example of CSSA form

Applying the concepts of CSSA in our test scenario in the previous chapter, the mapping process will transform relevant BPEL activities into CSSA forms like figure 15 above. At the end of one condition behavior, the out-going variable \( x_3 \) is
presented with a $\Phi$-function, that takes $x_1$ and $x_2$ as its possible values, whereby $x_1$ and $x_2$ are information the BPEL receives from a partner service. They are the same BPEL’s variable but have different subscript versions.

**Limitations and comparisons**

The CSSA has catered to its own needs to check for uninitialized variable access by considering some variables $v_0$ as initialized, but it does not cater to our own which requires specifying the matching elements between web services. There are two main differences in our solution concepts that allow it to handle better. First, instead of using a $\Phi$-function, the matching algorithm in chapter 5 generates all possible writers for a variable. Every possible writer (every match in our concept) is associated with a condition ID to “mark” that this writer is used under a certain condition (if there are any). And when there are conflicts between them (when there are more than one possible writers for one variable), the decision of taking the right writer for a variable depends on the runtime conditional information. It is important to know exactly the right writer for a variable in runtime (which partner service is chosen and writes on the variable), and then its associated UI annotations can be displayed to the UI correctly.

However, the CSSA approach does not deal with complex types of a variable. It is the reason why it cannot specify the concrete elements for the matches. Instead of using a $\pi$-function for the value of a concurrent variable in parallel process, our matching concept takes a deeper look inside a BPEL’s variable. In fact, although the BPEL’s variable can be defined from WSDL message, a XML Schema element or XML data type, it is still a set of elements. And the concurrent writer on a variable in a parallel process is just a writer on different elements of the variable. Hence, a writer on a variable in a parallel process is a merge of data in the BPEL’s variable in element details. Our matching concept considers a write on a variable in 3 cases: an equivalent, a sub set and a super set of elements in the BPEL’s binding variable. By doing this, it can specify correctly which parameter elements from which web service are matched with each other.

### 4.3 Deriving Explicit Data Links in WS-BPEL

**Description**

The algorithm presented in [34] from the University of Stuttgart, Germany, covers the BPEL process mostly as if it would perform in runtime and generate possible writers for variable elements (element in the BPEL’s binding variable). By analyzing the XPath expression in the condition behavior, the algorithm can over-approximate the behavior of the process and determine if a following writer on the running path would overwrite the previous writer on the same variable or not. If the analysis is well-done, hence the approximation, the algorithm can eliminate the previous writer (since it is overwritten) and return a more precise result. The whole
The technical report is described in details on [35]. The following part only summarizes the algorithm at some important points.

a. How the analysis covers the BPEL process?

The algorithm starts at the BPEL’s process element and analyzes each activity in its container. Analysis of a BPEL activity can only begin when all analysis of its predecessors have already been finished (by using a visited flag), if not, the analysis on this activity will be held off. When the analysis on an activity has been done, it continues to analyze successors to the activity (if the activity is part of a sequence), or goes to the outgoing BPEL links and jumps to the target activity (if there are any). For this to work correctly, the algorithm has to implement data handling for all BPEL structured activities. In [34], parallelization (flow activity) and branching (if-else activity) are handled by traversing activities separately. The separate data flows with separate possible writers for variables are then merged together. In the end a collection of all possible writers for variables are presented.

b. Analysis of a BPEL’s activity

The purpose of the analysis on an activity is to determine the possible writers to a variable. A writer to a variable is stated to 3 statuses: possible, disable and invalid.

If an activity writes to a variable (by using assign, invoke, receive, and rely), it will be added to the list of possible writers.

If one activity writes after another, the first writer will be moved to the invalid set (because it is overwritten by the second one). If, however, the second writer is under a conditional behavior (the activity might not be processed in runtime) and writes after the previous activity, the previous writer will be moved to the disabled set (instead of invalid set). Disabled writers will be re-enabled in all situations where the path concludes at dead ends.

Example

![Diagram](attachment:image)

Figure 16: Example of possible writers for variable x

Applying these concepts in our test scenario, the FlightBS (w1) writes flight information on variable x by default. Assuming variable x is the destination airport.
If the customer wants to rent a car (presented by condition \( c_1 \)), the AutoBS \((w_2)\) is activated and writes information on the same variable \( x \), which indicates the place to rent a car for an AutoBS. Following conclusions are returned from the algorithm.

- When encountering the conditional branch, the process will turn the first writer \( w_1 \) from possible over to disable writer, making the second writer \( w_2 \) the possible writer for variable \( x \). In runtime, if the condition \( c_1 \) is true, \( w_2 \) remains as the only possible writer for variable \( x \). If not, \( w_1 \) is enabled from disable to possible to replace the writer role taken by \( w_2 \).

- If the algorithm determines that the condition \( c_1 \) is always true in runtime (or there are no conditions between \( w_1 \) and \( w_2 \)), \( w_1 \) will be moved to invalid forever. And \( w_2 \) becomes the only one possible writer for variable \( x \).

- The algorithm can even gain more information: if \( c_1 \) is false in runtime and the join condition (JC) is an AND, the reading activity on variable \( x \) will be dead. It is interesting for compiler analysis but not useful for our objectives.

Limitations and comparisons

a. Structured activities

The analysis algorithm depends heavily on the structure of the BPEL activity. It has to cover the BPEL process as the process would be performed in runtime and thus the algorithm must handle all the behavior of the BPEL activity during implementation. In [34] only while, flow and scope are considered. The limitation of static analysis makes event activities more difficult to analyze, because it is statically undecidable if an event (onAlarm, onMessage, fault and termination) occurs at a certain activity or not at all. Randomly occurring events can disrupt the order of activities in the BPEL, making it impossible to be ascertained.

For our own needs, instead of following the process order, the matching algorithm generates all possible writers for a variable independent from the BPEL’s structure, and then a conflicts handler is used either to choose the correct writer for the variable or to merge the conflicts between them in the details of UI annotation.

In [34] the algorithm considers each assign activity for copying data between elements as a new possible writer of the variable. Due to the problem of the chain of data flow (section 3.3.1), our algorithm of “temporal matching” (section 5.3.2) considers each assign activity as a copy node (not a new writer) and determines the relation between the original source (from which previous writer activity the data actually comes from) and the target element in the assign activity. An activity to assign a literal value for a variable in the BPEL is irrelevant for the matching concept, since the initiated literal value does not have UI annotation.
b. An over-approximation on XPath expression

The task of enabling and disabling a writer belongs to an over-approximation, because it depends on the analysis of the XPath expression in the conditional behavior. If it is decidable, whether a condition is always true or not, it will create a good effect by reducing the possible writers on a variable. But it is shown and clearly stated in [29] that it is undecidable whether a given XPath expression is satisfiable or not.

For our needs, instead of “guessing” the conditional value in runtime, an “assertion” is inserted under the conditional branch. This “assertion” collects the value if the branch is processed by the BPEL process in runtime or not, and then return this information to the client. Depending on this value, the conflicts handler knows which branch is active in runtime. To say it in other words, it knows exactly which writer of the active branch is the correct writer for a variable. But gaining information from runtime by this way is not a static analysis anymore.

c. Limitations

Not following the sequenced order of activities may result in a case when two writer \( w_1 \) and \( w_2 \) both come into sequence at the same time (\( w_2 \) overwrites \( w_1 \)). Our conflict handler can decide which one to choose, so long as one of them has already been stated to be under a certain condition. But if they have no conditions attached, the matching algorithm will give out two possible writers, while the data analysis in this section gives only one writer \( w_2 \). Up to now the solution for undecidable conflicts is the merge of UI annotations.

Summary

The CSSA supports neither the complex type variable nor the BPEL condition. It only presents the dependence between variables with the \( \Phi \)-function for writing on the same variable and those with the \( \pi \)-function for concurrent value. The 2nd mechanism of explicit data links uses XPath expression analysis to over-approximate and reduce the number of possible writers where possible. Our algorithm inserts an assertion under the branch it interests instead, to determine the writer precisely. Furthermore the matching algorithm supports concluding the “front-end” relation between parameter elements of web services.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Complex type</th>
<th>Handling conditional branch</th>
<th>Web services relation</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSSA</td>
<td>none</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>Explicit Data Links</td>
<td>✓</td>
<td>Over-approximation</td>
<td>none</td>
</tr>
<tr>
<td>Our algorithm</td>
<td>✓</td>
<td>Assertion for accurate information</td>
<td>✓</td>
</tr>
</tbody>
</table>

Summary table: comparison between algorithms
5. SOLUTION CONCEPTS

In chapter 2 the ServFace project is introduced as an approach for the annotation of services with UI related information. Until now the annotation for a single web service is created with the help of an authoring tool. This chapter develops solution concepts, which automatically generate the UI annotation for the client (the composed web service) under the description of the BPEL.

The requirements analyses in chapter 3 are highly recommended before continuing with the following concepts. Readers may also refer to 2.2 for a summary of relevant information from the BPEL specification.

5.1 Algorithm Fundamentals

5.1.1 Serialization Process

The whole solution concepts developed in this paper describe a process, which takes the BPEL and the WSDL document of the client web service, the WSDL and the UI annotation document of all partner services as inputs. The process will then generate the UI annotation document for the client automatically. In some conflict situations between UI annotations in the client annotation, the BPEL and the WSDL of the client will be modified, so that the BPEL engine will return necessary runtime information for handling conflicts. The serialization of the process is illustrated in the following figure:

![Serialization process diagram]

Figure 17: Serialization process

5.1.2 Solution concepts overview

The whole concepts are divided in 4 main parts (figure 18 on the next page): the first part transforms all relevant activities from the BPEL for the data flow analysis with a node model. By using this node model as an input, a matching process generates all matches between parameter elements of the partner and the composed web service. Depending on the matching result, the third part presents concepts to copy the UI annotations from the partner to the composed web service. During the copy process, conflicts between UI annotations for the same parameter element may occur and it is solved by a conflicts handler. In some conflict situations, the UI annotation will be associated with some conditions to mark when it is used in runtime and when it is not in order to avoid conflicts. The decision for the right UI annotation depends on the runtime value the BPEL engine returns in runtime.
The final part presents concepts to modify the BPEL document, so that the BPEL engine will return runtime values which are required for handling conflicts.

Figure 18: Solution concepts overview

a. **A node model for matching algorithm** (section 5.2)

The requirements analysis in chapter 3 described the problem which occurs due to the chain of data flow (section 3.3.1). In the BPEL, data is written to, read from and transferred from element to element between the BPEL’s variables. There are different kinds and various forms of data assignment in the BPEL specification. To solve this problem and to facilitate the matching algorithm, a “node” presentation form is defined for all relevant activities (write, read, and copy activities) in the data flow.

b. **The matching algorithm** (section 5.3)

Taking this presentation model as an input, the algorithm will create the matching list of all possible parameter elements which are semantically equivalent with each other. A match indicates which parameter elements from the client are used for the partner, or which parameter elements from the partner service are the outputs to the client.

Figure 19: Transparent solution for the BPEL’s structured activities.

The requirement analysis in chapter 3 described about the problem of structured activities in the BPEL (section 3.3.2). To solve this problem, the matching algorithm is developed with a transparent property with the BPEL’s structure: it will consider each relevant activity in the data flow as an “independent node”. The
algorithm automatically determines the relation between these nodes and this process is independent from the BPEL’s structured activities. The nodes co-related by the algorithm are called relevant nodes.

For example, a write node writes data from the client in an element, and in a read node the data is sent to a partner’s service. These two nodes are relevant with each other. And the data relation between these two nodes is a matching result of the matching algorithm.

![Figure 20: transitive calculation. Find out relation between 1, 2 and then 3.](image)

Due to the potential problem that could occur in the chain of data flow, the relation between two relevant nodes is not always the relation between the partner and the client. There may be intermediate copy nodes between them, which specify the internal data flows inside the BPEL engine. For this reason, a concept for “temporal matching” is developed to calculate the transitive relation between an intermediate copy node and the root node (a write node where the data flows in the BPEL engine). The matching process continues up the node hierarchy until it reaches an end node, at which the relation between the partner and the client are defined.

Section 5.3.1 will describe the fundamentals of the matching algorithm again in details. Section 5.3.2 and 5.3.3 will develop concepts for finding relevant nodes from a single node and for the “temporal matching” respectively. In the end, section 5.3.4 will describe the matching process step by step with a sequence diagram.

c. **UI annotation & conflict handler (section 5.4 & 5.5)**

The outputs from the matching algorithm can indicate the matches between two messages, their parts or elements. An element in turn is annotated with a set of UI annotations. The copy operation of UI annotations has to search for a set of parameter elements from the partner service operation and copy their UI annotations to the matching elements of the composed web service.

When a conflict between two parameter elements occurs, their UI annotations are merged together (figure 21), resulting in a new set of UI annotations for the composed service’s element. But when the merged set has two or more UI annotations from the same type, one of them has to be chosen for annotating the composed service’s element by default (a2 in the figure 21).
The concept in section 5.5 deals with several situations for a default annotation. But in case the default annotation cannot be decided, the annotations are included with a condition id, which is assigned as follows.

When a node (a relevant activity) is under a branch of one condition behavior (if-else), the node is set with a condition id, which is the id of the branch. The matching algorithm will include all condition ids of every relevant node it could find on the way. So the final match collects a set of condition ids. This set of condition ids presents the path the BPEL process will go through in runtime as it encounters various branches. So if the conflict annotations are included with this condition attribute, they can be differentiated from each other. When a conflict annotation has its conditions in runtime satisfied, it will be used in the client GUI.

The concept for UI generation is first presented without taking the possibility of conflicts into consideration (section 5.4) and hence is later extended with a conflicts handler (section 5.5).

d. The BPEL modifier (section 5.6)

To gain the value of conditions in runtime, the BPEL modifier inserts an “assertion” under the condition enclosing element in every branch it interests, so that when the BPEL process accesses this branch, the assertion is set (true). The set of assertions are then embedded to the output variable and replied to the client. Depending on the value of the assertion, the annotation with satisfied conditions will be used in the client GUI.

5.2 A node model for matching algorithm
In the BPEL, relevant activities for the data flow analysis are activities which perform an operation on the data. The data here are the BPEL’s variables or the elements inside the BPEL’s variables. Write activities, such as receive and invoke (when invoke activity receives a response), will write data on the variable. Read activities, such as reply and invoke (when invoke activity send a request), read the data from the variable. There are copy activities which read the data from an element inside a variable and then copy it to another one. These write, read and copy activities take an important role in the data flow. This section aims to transform them to a node presentation for the matching algorithm.

a. A presentation form for data selection

An activity is considered as a node with an attribute selection, which directs the activity to a certain data to operate with. The selection is presented in a 3-tuple \{variable, part, XPath\} form with:

1. Variable binds to the BPEL’s variable. A BPEL’s variable can be defined from a WSDL messageType, XSD element or XSD data type.
2. Part is a part of a message in the BPEL’s variable, if the variable is from messageType. If not, it can be left as null.
3. XPath is the absolute location path of an element inside the binding variable.

In the BPEL specification, an activity can perform an operation on a data with various possibilities: it can operate on the whole WSDL message or a single part of the message, or a single XML information item such as EII and AII in the binding variable (8.4.1) [3]. With the above 3-tuple form, it can thus present all possibilities of a data selection. For example:

1. Selection of a BPEL’s variable v: \{v, null, null\}
2. Selection of a part of the message p in a variable v: \{v, p, null\}
3. Selection of an element e inside a part of a message p of the variable v from messageType: \{v, p, e\}.
4. Selection of an element e inside a variable v from type element: \{v, null, e\}.

b. Advantages

The presentation form looks simple and it is necessarily so, because the BPEL specification, for the data selection, has various forms, all of which must be transformed into one united presentation form. This section will not delve further into the details of this transformation. The transformation, for your information, is described on “handling several forms of assignment” (section 6.2) of the implementation chapter.
The most important advantage of this presentation form belongs to the ability to find out the relationship between two data selections. For example, let’s consider two selections below:

\{\$v, null, null\} and \{\$v, p, e\}

If variable \(v\) of the two selections is the same, the second selection is definitely a subset element of the first one, since the second selection references to an under element \(e\) inside the variable \(v\). The 3-tuple presentation for a data selection is designed so that it will be useful for the concept of finding relevant nodes among independent nodes (section 5.3.2).

\[\text{c. A node presentation for activity}\]

\[
\begin{array}{c}
\text{W} \\
\{\text{variable1, null, null}\} \\
\text{FROM} \\
\{\text{variable1, part1, xPath1}\} \\
\{\text{variable2, part2, xPath2}\} \\
\end{array}
\]

Figure 23: a write node (above) and a copy node (below).

Write and read activity operate only on a BPEL’s variable. They are presented by nodes with one selection attribute \{\$variable, null, null\}, whereby part and xPath in the tuple are always null.

The copy activity can copy a BPEL’s variable, or a part of a message or an element inside a variable to another one. It is presented by a node which has two selection attributes: selection FROM and selection TO.

\{\$variable1, part1, xPath1\} \rightarrow \{\$variable2, part2, xPath2\}

//For example, an element "street" in part "address" of message-
//typed variable "person" is copied another element "road", which
//is a child element of element-typed variable "location".

//The copy activity is now presented with:
\{\$person, address, street\} \rightarrow \{\$location, null, road\}
5.3 The matching algorithm
5.3.1 Matching algorithm fundamentals

![Diagram of tree data flow]

**Description of the tree:** now imagine there is a hierarchal tree with a set of linked nodes. Each node is one relevant activity. Write nodes are on the top of the tree. Read nodes are at the bottom of the tree. Between write and read nodes are copy nodes (intermediate nodes). The tree represents the data flow inside the BPEL, when nodes are inter-connected. Data is written in the BPEL at a write node. The data is then copied between copy nodes and in the end it is read by a read node.

**Goal:** the matching algorithm has to find out the relation between every write node (at the top of the tree) and every read node (at the bottom of the tree). However, the matching algorithm has to overcome the intermediate nodes (between the top of the tree and its bottom) to find out the final relation which, as a matter of fact, is not always directly linked.

**Concepts:** the matching algorithm considers each node independently. In the following section 5.3.2, a concept will be developed to determine which independent nodes are relevant to each other. In general, they are nodes with relevant data selection. Nodes are considered irrelevant nodes if they are not linked from root, directly or indirectly.

In section 5.3.3, a concept of temporal matching will be developed as follows. When a node finds a new relevant node, the temporal matching will calculate a data relation between the very 1st node (root node) and the new relevant node. A new connection is hence established.

The whole picture is very clear now. The matching process starts from a root node and finds all relevant nodes to it. For every relevant node that the matching process
finds out on the way, it establishes a temporal match with the root. When a node cannot find any more relevant nodes, it is considered as a terminal node. The result is a dead-end path. Afterwards, the matching process navigates up by one level in the node tree and continues to find other relevant nodes to the one-level-up node it has just encountered. For example, if node (4) in the figure 24 is a terminal node, the process will go back to node (3) and continues to find a relevant node (5).

During the run, any temporal match which involves at least a read node will be added to the final matching list. Since the read nodes are always at the bottom of the tree, they are called final nodes particularly during the matching process.

As a result of the entire process discussed at length as above, the final matching list represents all relation between the web service of the client and that of his partners in details. The presentation form of a match can specify which message, part of a message, or elements in the client operation are matched to which ones in an operation of the partner’s web service accordingly.

a. Two types of run

![Diagram](image)

**Figure 25: type 1, from BPEL’s input to Partner’s input**

The matching process has two types of run: (1) “from BPEL’s input to Partner’s input” and (2) “from Partner’s output to BPEL’s output”. Type (1) will find out the relation between the input of the client operations and the input of its partner operations. In this direction, the data that the BPEL receives from the client in an operation is written to a variable. The data is copied and in the end invoked in partners’ operations. The matching process will begin with each write node of the client as the start root node and ends at a read node of the partner as the final node.

![Diagram](image)

**Figure 26: type 2, from Partner’s output to BPEL’s output.**
The second running type is “from Partner’s output to BPEL’s output”. This type of run will find out the relation between the output of the partner’s WS operations and the output of the client operations. In this direction, the data that the BPEL receives from the partner in an operation is written back to a BPEL’s variable. The data is then copied and replied to the client. The matching process will begin with each write node of the partner as the start root node and ends at a read node of the client as the final node. Although the two types of run only have different inputs and outputs, the algorithm is the same.

b. What is temporal match backup?

![Diagram](image)

Figure 27: the calculation of new temporal match (1-5) requires previous match (1-3)

In the figure 27, readers may notice that there is a backup temp. What is it for? Imagine that the process is at node (3) of the tree, and it has just found a relevant node (4). The temporal matching will be immediately updated with a new relation between node (1) and (4). The process then goes down to node (4), which is a terminal node, as there exists no relevant nodes to it. Afterwards the process returns to the previous node (3), and searches about if there’re still other relevant nodes of (3). If other relevant nodes exist, the process will calculate the new temporal match accordingly.

The calculation of new temporal match between node (1) and (5) (section 5.3.3) requires the previous temporal match between (1) and (3), but it is missing (the current temporal match is about a relation between node 1 and 4). Thus the previous temporal match must be backed up for the use in this case.

In short, a temporal backup is “cloned” from the current temporal match one at a time, whenever the process identifies a new relevant node to the current node. And the more relevant nodes the process could find at one node, the deeper it would go into the node hierarchy branched from that node. But when it bounces back from the terminal nodes, the backup temp is already established for calculating the new temporal match.
c. Dead-lock preventing

![Diagram](image)

**Figure 28: endless run is not allowed**

For some reasons the developer of the BPEL’s process writes some data assignments either in a parallel processes or in an IF condition, for example:

If $a < b$: assign $a \rightarrow b$
else assign $b \rightarrow a$

… by which he does not imply meaningless assignments. The seemingly redundant assignments are used to make up for what the matching algorithm does not take into account. The matching algorithm does not consider if the BPEL’s process is sequential or parallel or if it has IF condition. In fact, it only considers which copy nodes in a list are relevant and then performs the calculation of temporal match.

With the example above (at figure 28), the algorithm only sees that it has 2 copy nodes in the list: $a \rightarrow b$ and $b \rightarrow a$. Both are relevant with each other. So an endless run between the two nodes occurs, when the algorithm starts. During the run it always finds a relevant node to the current node and thus the matching process never ends. Although this case is very rare, an endless run is not allowed for a complete algorithm. The following is a solution to avoid the endless run.

The solution idea comes from the tracing collectors "mark and sweep" of the Java virtual machine [9], whereby in the mark phase, the garbage collector marks each object it encounters; and in the sweep phase, unmarked objects are freed, and the memory freed up is made available to the executing program.

Our solution is that the matching process “marks” every node that it encounters, so that it doesn’t read that node again until the node is released. The node is released and is considered a terminal node when the process cannot find any relevant nodes to. Every copy node is tagged with an id, so that the process can “mark” a tagged node as currently “on board”. In particular, when the 2nd node is considered as relevant to the 1st node, the ID of 2nd node will be added to a “being-considered list”. The id of this node will be later removed from the list as and when that node is considered as a terminal node. Here comes with an example:
Figure 29: Endless run prevention

The figure above clarifies the situation when (1), (2), (3) and (4) are currently “on board”; the matching process cannot access any of these nodes. On the figure, node (4) and node (3) are presumably “terminal nodes”. Both node (4) and (3) are sequentially removed from the “being-considered list”. Thus the process moves back to node (2), as a result. Since node (2) has node (5) as its relevant node, the path at 1-2-5 is created. In the later period, the process can access node (3) or (4) again, only if they are relevant to node (5).

Matching algorithm summary

1. When the algorithm begins?
For running type 1 “from BPEL input to Partner input”, it begins with each write node of the client. For running type 2 “from Partner output to BPEL output”, it begins with each write node of the partner.

2. When will the algorithm terminate?
A run starts from a root node to a terminal node. A terminal node is a node which cannot find any further relevant nodes to it, resulting in a dead-end path. The process goes back to the previous one-level-up of the tree. The whole algorithm terminates when all runs terminate, which means when all direct and indirect relevant nodes to root have been considered.

3. What is a temporal match?
A temporal match is the relation between one root node and intermediate node. It is calculated by the table rules in section 5.3.3, for every time the process finds a new relevant node.

4. What is the final matching list? Final node?
During the process if the current temporal match is related to a final node, the temporal match will be added to the final matching list. Final nodes are usually read nodes. They are partner’s read nodes for running type 1 or client’s read nodes for running type 2. So the final matching list contains only the temporal matches. Each of them specifies which message, part of a message, or elements in the client operation are matched to which ones in an operation of the partner’s web service accordingly.
5.3.2 Concept for finding relevant nodes

In the previous section, every activity is treated as an independent node in the beginning. A concept has to be developed into the matching algorithm in order to find the relation between these independent nodes. In this section, we aim to develop a concept which starts with any node on the tree and finds out all the relevant nodes among the rest. The following examples are useful for the definition of two relevant nodes.

a. Examples

The first example illustrates the relation between a write node and a copy node. In the 1st write node, data received from the client is written to the variable `Travel`. In the 2nd copy node, an element `date` in the variable `Travel` is copied to another element in the variable `Flight`. These two nodes are relevant because the selection FROM of the 2nd node (element `date`) is a “sub set” element of the selection of the 1st node (variable `Travel`).

![Figure 30: a write node finds its relevant node due to the FROM selection.](image)

The second example illustrates the relation between two copy nodes. In the 1st copy node, the data is copied to an element `date` in the variable `Flight`. In the 2nd node, the whole variable `Flight` is copied to variable `Auto`. These two nodes are relevant because the selection FROM of the 2nd node (variable `Flight`) is a “super set” element of the selection TO of the 1st node (element `date` in the variable `Flight`).

![Figure 31: a copy node (above) finds its relevant node (below).](image)

b. Definitions for relevant nodes

Two nodes are relevant, when the data selections in them are “relevant”. For 2 copy nodes, the selection TO of the 1st copy node is relevant with the selection FROM of the 2nd copy node.
It is always important to define which and when two data selections are considered as ‘relevant’. The following will deal with this concern.

c. Categories of relevant nodes

There are 3 possibilities for two selections to be relevant: (1) they reference to the same data (their 3-tuple presentation forms are the same), (2) one of the two selections is a sub set element or (3) a super set element of the other. The table below organizes the relationship between two data selections and arranges them in 3 categories: equivalent, superset and sub set.

<table>
<thead>
<tr>
<th>1st TO</th>
<th>2nd FROM</th>
<th>$v1</th>
<th>$v1, part1</th>
<th>$v1, part1, x1</th>
<th>$v1, null, x1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v1</td>
<td>equivalent</td>
<td>sub set</td>
<td>sub set</td>
<td>sub set</td>
<td></td>
</tr>
<tr>
<td>$v1, part1</td>
<td>super set</td>
<td>equivalent</td>
<td>sub set</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$v1, part1, x1</td>
<td>super set</td>
<td>super set</td>
<td>equivalent</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$v1, null , x1</td>
<td>super set</td>
<td></td>
<td>equivalent</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Filter table for relevant nodes

⚠️ In the table, 4 cells are left blank, because these cases never occur. Please refer to the end of this section for an explanation. Readers may be confused at first. But don’t worry about that. Each category will be discussed right now.

(1) Equivalent

Two selections are equivalent when they reference to the same data. Their 3-tuple presentation forms are exactly the same. The example below demonstrates the relation between 2 copy nodes, when their data selections are the same.

FROM                  TO
1. {...}              ➔  {${v1, part1}
2. {${v1, part1}}    ➔  {...}

// The 2nd node has the selection FROM equals to selection TO of // the 1st one.

(2) Selection of the 2nd node is a sub set element of the 1st

One selection is a sub set element of the other when it references to a child element, or a part of a message, whereby the other selection references to a parent element or the entire WSDL message respectively.

FROM                  TO
1. {...}              ➔  {${v1, part1, null}
2. {${v1, part1, x1}} ➔  {...}

// The 2nd “from” references to an element x1 // The 1st “to” references to a message’s part, containing x1.
(3) Selection of the 2nd is super set element of 1st

This category is the reverse case of the second one. One selection is considered as a super set element of the other when it references to a parent element or a WSDL message, whereby the other selection is a child element or a part of the WSDL message respectively.

FROM                      TO
1. {...}                  \rightarrow  {v1, part1, x1}
2. {v1, part1, null}       \rightarrow  {...}

// The 2nd “from” references to a message’s part.
// The 1st “to” references to a child element x1 in message’s part

d. Impossible relation

In the table above, 4 cells are left blank as these relations could never occur. Let’s take out one case for an example:

{$v1, \text{null}, x1}$ & {$v1, \text{part1}, \text{null}$}

The 1st selection references to an element x1 inside the variable v1. Since the part of the 1st selection is null (there are no specified parts of a message to contain the element x1), the container variable v1 for the element x1 has to be from type element or XML Schema type, but definitely not a messageType. In the 2nd selection, the variable v1 must be a messageType, because the variable v1 has a message’s part part1 inside of it. The variable v1 cannot be both an element type and a messageType; in the end, its type compatibility conflicts, so this situation of relation is impossible.

e. Irrelevant relations

Two selections are irrelevant, for example, when they reference to two different variables, two different parts of the message-typed variable or two different elements in the variable. For example:

FROM                      TO
1. {...}                  \rightarrow  {v1, part1, x1}
2. {v1, part1, x2}         \rightarrow  {...}

// the 2nd “from” is about an element “x2” whereby the 1st “to” is // about an element “x1”.

In the example above, although the two elements are in the same part of the message, data flow in the element x2 will not affect x1. These two elements can also have different UI related annotation descriptions, so they are not necessarily relevant with each other.
Summary

In this section, a concept is developed to collect relevant nodes from a starting node. Two nodes are relevant when their data selections are relevant. Two copy nodes are relevant when the selection FROM of the 2nd node is relevant with the selection TO of the 1st node. Two data selections are relevant when one data selection is equivalent, a super set or a sub set element of the other one.

5.3.3 Temporal matching

![Diagram](image)

Figure 32: (1), (2) and (3) are temporal matching. (a), (b) and (c) are relation between nodes.

In the previous section, a concept is developed to find out relevant nodes from any one node (relation a, b, c in the figure 32). In this section another concept is developed for concluding the “temporal match” (relation 1, 2, 3). Apart from the relations between relevant nodes as discussed previously, this temporal match defines another type of relation - one between the very first start node and any intermediate nodes along the path from that start node to the final node, as illustrated on figure 32.

a. Definition of temporal matching

![Diagram](image)

Figure 33: a full transitive conclusion

Let’s assume there are 2 relations a→b and b→c, a mathematical transitive conclusion [5] should yield a new relation a→c. As a result, two given relations are combined and then reduced into a new one. The new relation created out of the old two is called a temporal match.

💡 When any two copy nodes are relevant to each other, a temporal match creates a relation between the selection FROM of the 1st copy node and the selection TO of the 2nd copy node, at first. Then, If the process finds any 3rd copy node as relevant to the 2nd copy node, temporal matching creates a new relation between the selection FROM of the 1st node and the selection TO of the 3rd copy node. And so on when more relevant nodes to the 3rd are identified.
The end-result of the concept is a reduction of a series of relations possible between intermediate nodes into one relation, which is between the very 1st node and the end node.

Figure 34: “temporal matching” is a relation between the 1st “from” & the 2nd “to” attribute.

b. Table rules for temporal matching

When the selection FROM of the 2nd node and the selection TO of the 1st are equivalent (a→b and b→c), the temporal matching between the two is considered a full transitive conclusion (a→c). But the relation between two nodes is not always a full transitive conclusion like the case above. There are also super set and sub set relations. Apparently, how the temporal match is calculated for all possible cases can be complicated and confusing. A rule is established in the table below to define the calculations:

<table>
<thead>
<tr>
<th>1st FROM</th>
<th>1st TO</th>
<th>2nd FROM</th>
<th>2nd TO</th>
<th>temp FROM</th>
<th>temp TO</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equivalent</td>
<td>*</td>
<td>$v1$</td>
<td>$v1$</td>
<td>unchanged</td>
<td>2nd TO</td>
</tr>
<tr>
<td>*</td>
<td>$v1$, $p1$</td>
<td>$v1$, $p1$</td>
<td>$v2$</td>
<td>unchanged</td>
<td>$v2$, $p1$</td>
</tr>
<tr>
<td>*</td>
<td>$v1$, $p1$, $x1$</td>
<td>$v1$, $p1$, $x1$</td>
<td>$v2$, $null$, ($x$)</td>
<td>unchanged</td>
<td>$v2$, $p1$, $x1$</td>
</tr>
<tr>
<td>*</td>
<td>$v1$, $null$, $x1$</td>
<td>$v1$, $null$, $x1$</td>
<td>$v2$, $p2$, ($x$)</td>
<td>unchanged</td>
<td>$v2$, $null$, ($x$) $x1$</td>
</tr>
<tr>
<td>*</td>
<td>$v1$, $p1$, $x1$</td>
<td>$v1$, $p1$, $x1$</td>
<td>$v0$, $p1$</td>
<td>*</td>
<td>$v0$, $p1$, $x0$</td>
</tr>
<tr>
<td>*</td>
<td>$v0$, $null$, (x)</td>
<td>$v0$, $null$, (x)</td>
<td>$v0$, $null$, (x) $x0$</td>
<td>unchanged</td>
<td>$v0$, $null$, (x) $x0$</td>
</tr>
<tr>
<td>*</td>
<td>$v0$, $p0$, (x)</td>
<td>$v0$, $p0$, (x)</td>
<td>$v0$, $p0$, (x) $x0$</td>
<td>unchanged</td>
<td>$v0$, $p0$, (x) $x0$</td>
</tr>
</tbody>
</table>

Please be reminded that the conclusion of temporal match only happens for two relevant copy nodes. The 2nd column (1st TO) and the 3rd column (2nd FROM) of the table list all variants of two relevant copy nodes known from the previous section. They are arranged in 3 categories: equivalent, super set and sub set.

The mathematical property of set elements will have implications on any new temporal match. For your convenience, each category of the relation is explained with examples as follows.
(1) Equivalent

Sometimes events in real life help us to understand the situation better than mathematical symbols. There are 2 statements as follows: “John gives me a candy” (a→b) and “I give it to Marry” (b→c). What is concluded? The answer will be “John gives a candy to Marry indirectly” (a→c).

In this case, it is a full transitive conclusion (both the 1st TO and the 2nd FROM are the equivalent). The new temporal matching will keep the selection FROM of the 1st node as its selection FROM (a), and update its selection TO with the selection TO of the 2nd node (c). The figure 33 on the previous page has already illustrated this case very clearly.

(2) The 2nd FROM is a super set element of the 1st TO

Now return to the story. There are 2 statements as follows: “John gives me a candy” and “I put it in a box and give the box to Marry”. The answer has to be “John indirectly gives Mary a candy, which is in a box”. The answer for the two relevant nodes of this category is the same:

When element x0 is copied to element x1, but a superset of this element (variable v1) is copied to another superset (variable v2), x0 is indirectly copied to x1 in v2 as a result.

![Figure 35: “x0” is indirectly copied to “x1” in superset V2.](image)

In general, our temporal matching will keep its selection FROM unchanged, for whatever it is in the beginning. But the selection TO of the temporal matching will be updated with x1. The sub set x1 can be either an element inside a parent element, an element inside a part, or a part of a message, as long as the super set element v2 is a parent element, a part or the entire message respectively. Readers may skip the following parts as they only explain the 3 cases possible under this category in details with its logical implication.

Superset v2 is message type.

```
FROM     TO
// Data is copied to an element x1 inside a message v1
1. {...}  -> {v1, part1, x1}
```
Superset \(v^2\) is message’s part

FROM \(v_1\) TO \(v_2\)
// Data is copied to an element \(x_1\) inside a message’s part1
1. \{…\} \(\rightarrow\) \{$v_1, \text{part1}, x_1\}

// message’s part1 is copied to another message’s part2.
// Both part MUST be from the same element, or the same data type.
2. \{$v_1, \text{part1}\} \rightarrow \{$v_2, \text{part2}, (x)\}

// Temp matching takes the selection FROM of the 1st node and the
// selection TO references to an element \(x_1\) inside message’s part2
// of message \(v_2\) (not to the whole part2).
\{…\} \(\rightarrow\) \{$v_2, \text{part2}, (x)x_1\}
*x is an optional element, which declares a child element of
message’s part is in the relation, not the message’s part.

Superset \(v^2\) is an element

FROM \(v_1\) TO
// Data is copied to an element \(x_1\) inside a parent element \(v_1\).
1. \{…\} \(\rightarrow\) \{$v_1, \text{null}, x_1\}

// parent element \(v_1\) is copied to another element \(v_2\).
2. \{$v_1\} \rightarrow \{$v_2, \text{null}, (x)\}

// new relation is about element \(x_1\) inside \(v_2\).
\{…\} \(\rightarrow\) \{$v_2, \text{null}, (x)x_1\}
*x is an optional element, which declares a child element of the
element \(v_2\) is in the relation, not the \(v_2\) itself.

(3) The 2\textsuperscript{nd} FROM is a sub set element of the 1\textsuperscript{st} TO

Event in real life with 2 statements: “John gives me a box which contains a candy”
“I give the candy to Marry”. The new conclusion has to be “A candy in John’s box
is given to Marry”.

💡 When a superset \(v^0\) is copied to a superset \(v_1\), but only a sub element \(x^0\) of this
superset \(v_1\) is copied to \(x_1\), the corresponding mapping element of \(x^0\) in \(v^0\) is copied to
\(x_1\) indirectly as a result.
Figure 36: only an element “x0” of “v0” is indirectly copied to “x1”, not the whole superset.

In general, the temporal matching will keep its selection TO unchanged, for whatever it is. But the selection FROM of the temporal matching will be updated with x0, which is either a child element of an element, an element of a part or a part of a message, if the superset v0 is a parent element, a part, or the entire message respectively. Readers can skip the following parts as they only explain the 3 cases possible under this category in details with its logical implication.

**Superset v0 is a message type**

// a message v0 is copied to another message v1.
// Both v0 and v1 MUST be from the same messageType
1. {v0} → {v1}
// only an element x0 in message v1 is copied to another (to
// whatever but it has to be an XML information item set, which is
// a type compatible with x0)
2. {v1, part1, x0} → {...}

// temp matching takes the selection TO of the 2nd copy node and
// the selection FROM references to an element x0 in message v0
{v0, part1, x0} → {...}

**Superset v0 is a message’s part**

// a message’s part0 is copied to another message’s part1
// Both MUST be from the same element or data type.
1. {v0, part0, (x)} → {v1, part1}

// only an element x0 in the message’s part1 of the
// message v1 is copied to another.
2. {v1, part1, x0} → {...}

// temp matching will take the selection TO of the 2nd copy node
// and the selection FROM references to an element x0 of part0 in
// message v0 (not to the whole part).
{v0, part0, (x)x0} → {...}

*x is an optional element, if an element inside the message’s part0 is in relation, not the whole message’s part itself.*
Superset v0 is an element

// variable v0 (a complex element) is copied to another variable v1
// Both MUST be from the same element or data type.
1. {$v0, null, (x)} → {$v1}

// only an element “x0” in v1 is copied to another.
2. {$v1, null, x0} → {...}

// temp matching will take the selection TO of the 2nd copy node
// and the selection FROM references to an element “x0” of v0 (not
// to the whole variable v0).
   {$v0, null, (x)x0} → {...}
*x is an optional element, if you want to specify an element inside
the element v0 is in relation, not the whole element v0 itself.

Summary: in this section a concept for temporal matching is developed. The
temporal match is a new relation with between the attribute FROM of one copy
node and the attribute TO of another. These two attributes present the new relation
between the selection FROM of the very 1st node and the selection TO of an
intermediate node. The relation is deemed true based on the logical implication
between two transitive sets of elements. All possible cases under temporal
matching are arranged in 3 categories for 2 relevant copy nodes: equivalent,
superset and subset.

The idea behind the temporal matching is to reduce the relations between all
intermediate copy nodes between the first and the end node into one. In this
manner, the temporal matching will help the matching algorithm overcome the
problem of the chain of data flow while the algorithm attempts to find out the data
relation between “front-end” parameter elements of the composed web service and
a partner web service.

5.3.4 Flowchart of the matching algorithm
The matching algorithm is described steps by steps as follows. Readers may refer
to the “matching algorithm fundamentals” (section 5.3.1), which introduces the
main idea behind the instruction steps.

Step 1 - Initiation: the 1st node is initiated with a start node, which is either a
client’s write node for running type 1 or a partner’s write node for running type 2.
The start node has a selection form: {$variable, null, null} presenting data
from external service is written to a BPEL’s variable.

The temporal match is initiated with the selection FROM and the selection TO,
the former of which references to the same variable of the start node while the
latter of which is still left blank:

{$variable, null, null} → {...}
Figure 37: Flowchart of the matching algorithm

Step 2: Find relevant nodes

FOR EACH copy nodes in the list, check if it is relevant with the 1st node, according to the table rules in section 5.3.2.

IF YES

- (2a) the ID of the newly found relevant node is added to a blocking list. The current temporal match is backed up. A new temporal match is then calculated according to the table rules in section 5.3.3.

Figure 38: find a relevant node in step 2. And mark the node as relevant in step 2a.
(3) Check if the current temporal match is related to a final node (in the fundamentals of matching algorithm, final node is a read node at the bottom of the tree). In details, for the running type 1 “from BPEL’s input to Partner’s input”, the process checks if the selection TO of the temporal match references to the same output variable of a partner’s read node. For the running type 2 “from Partner’s input to BPEL’s output”, the process checks if the selection TO of the temporal match references to the same output variable of a client’s read node.

- (3a) IF YES, the current temporal match is added to the final matching list (△) and go to step 4.
- (3b) IF NO, the current temporal match is not related with any final nodes and go to step 4.

(4) In this step the process goes down to the 2nd copy node (the relevant node just found, in other words). A new recursive run is started again beginning from step 1, but with a different 1st node and temporal match. The 2nd copy node and the temporal match of the last run are forwarded to the next run.

![Figure 39: Step (4) the 2nd copy node becomes the 1st node for the next run.](image)

IF NO relevant nodes are found:

(2b) The process continues to look for relevant copy nodes in the list, until no relevant copy nodes are found at all. The process will determine that this current 1st copy is a terminal node. Hence, the current 1st node’s ID is removed from the blocking list. Afterwards, the process restores the backup temporal match of the previous run, and meanwhile, terminates this recursive run and goes back to the previous run.

![Figure 40: at terminal node, go back to previous node & look for other relevant nodes](image)
The matching process terminates when all recursive runs terminate (all relevant nodes from root have been considered).

⚠️ In step (3a), when a temporal match is added to the final matching list, this match is also added with other information about the operation, portType and partnerLink of the start node and the final node (table below). This additional information allows the match to indicate to which operations and services the matching data selection belongs.

<table>
<thead>
<tr>
<th>BPEL's WS</th>
<th>PARTNER's WS</th>
</tr>
</thead>
<tbody>
<tr>
<td>(v_1, p_1, x_1) \ op1 partnerLink1</td>
<td>(v_2, p_2, x_2) \ op2 partnerLink2</td>
</tr>
</tbody>
</table>

Figure 41: final matching list with BPEL’s WS (left column) & Partner’s WS (right column)
5.4 Generation of UI annotation

Previously a matching algorithm is developed to generate the matches between a client and a partner operation in the following presentation form:

\[
\text{Client} \quad \{v_1, p_1, x_1\}, \text{op}, \text{plink} \rightarrow \{v_2, p_2, x_2\}, \text{op}, \text{plink}
\]

In addition, every match contains the information about the operation and partnerLink which allows the match to indicate to which operations and services the matching data selection belongs. The 3-tuple presents a data selection inside the corresponding operation. In the following section, depending on information from the matches, annotation elements (AE) from the partner service are copied to the client.

5.4.1 Goals

The data selection of a match can reference to a message, a part or an element of a part in an operation. In particular, it references to a set of parameter elements in an operation and each parameter element in turn has a set of UI annotation elements in the annotation document.

![Figure 42: Copy annotations from the partner’s parameter to the client’s parameter](image)

The figure above illustrates the task of the annotation generation: for every parameter element $e_2$ of the partner operation, which is specified by the match, copy all AEs of it to the matching parameter element $e_1$ of the client operation. The annotation generation is formalized as follows:

1. For each partner’s parameter element specified by the match
   $\forall e_2 \in E_2$ with $E_2 = \{v_2, p_2, x_2\}, \text{op}, \text{plink}_2$ 
2. Get all AEs of it
   $A(e_2) = \{a | a \text{ is an annotation for element } e_2\}$
3. And copy these AE for the annotation of the client parameter
   $A(e_1) = A(e_2), \forall e_1 \in E_1$ and $\forall e_2 \in E_2$
5.4.2 Requirements analysis
The following part describes how the copy process locates the correct annotations in the partner document with the given matching input.

Figure 43: Referencing Mechanism for parameter element and DTElem [2]

Under the Referencing Mechanism of (2.2) [2], there are two types of annotation: “annotation for a parameter element” of a service operation and “annotation for a data type element” (DTElem) of a local data type (LDT).

The process should copy both annotation types for the composed WS:

1. The “annotation for ParamElement” means that the annotations from parameter elements of the partner WS operation are copied to the parameter element of the composed WS operation.

2. The “annotation for LDT” means that the annotations from LDT of the partner WS are copied to the LDT of the composed WS. It happens when a matching parameter of the partner WS is defined from a data type and the partner WS has annotations for this data type.

The copy process depends on the matching information to find the correct annotations within the document. The difference between the two annotation types is only the Referencing Mechanism (2.2) [2]. This mechanism specifies each annotation referenced by a unique attribute hierarchicalName in the annotation document. The copy process only has to find the annotations with the correct hierarchicalName, regardless which type of annotation it is.

a. Annotation for ParamElement

The referencing mechanism for a parameter element requires the web service name, the operation, the ParamSet (input or output message), the WSDL part and the absolute location path to the parameter element inside the part.
The data selection of a match has the part and xpath referenced to the part and the absolute location of the parameter element in the hierarchicalName respectively. The partnerLink of the match also connects to the corresponding WSDL for querying the WS name. Operation and paramSet are already given by the match. So for a given match, all required information for referencing a parameter element in the annotation document is satisfied.

Problems

Problem occurs only when the data selection has the part information missing \{variable, null, xpath\}. It happens when the “whole” message of a web service is matched to another one (no concrete part of a message specified), or when the variable of the data selection is an element-typed variable (only message-typed variable of a BPEL operation contains parts. Element-typed variable has no such parts in itself, because it externally references directly to a single part of the message).

Solution

Problem 1: if it is a match between two messages, the process simply queries all parts in the message for the copy.

Problem 2: if the BPEL’s variable is an element-typed variable, the process has to query for the part in the WSDL, which is referenced by the element-typed variable. The query process is simple as follows (step by step):

1. Query portType in the WSDL belonging to the given partnerLink.
2. Query the message of the operation and paramSet in this portType (!).
3. Query the single part inside this message.

Figure 44: example hierarchicalName of parameter element startAirport

Figure 45: get message's part name from partnerLink, operation and paramSet
Operation names in WSDL are not unique (2.5) [4], so the query cannot locate an operation directly. Instead it has to find the unique operation in a portType first.

b. Annotation for DTElement

If a WSDL part or a parameter specified by the match is defined from a local data type, the annotations of this data type have to be searched and copy. The referencing mechanism for a LDT requires the name of the service and the LDT. Due to type compatibility, both matching parameters in the services must be defined from the same XSD data type, so the name of LDT in both services are the same and only their web service names’ are different.

5.4.3 Flowchart

The following part describes the UI generation process steps by steps.

FOR EACH match

Step 1: Check if the variable of the data selection is from the element type?

- (1a) if yes, find the part in WSDL referenced by the variable.
- (1b) if no, the variable is a messageType.

Step 2: Check if the data selection has a part specified?

- IF no
  - (2a) Query all parts in the message which are defined from XSD data type. For each of them, process step 4a.
  - (2b) Query all parts in the message, for each of them process 4b.
• IF yes
  o (3a) if the part is defined from XSD data type, process step 4a.
  o (3b) if the part is defined from XSD element, process step 4b.

Step 4: Copy annotation elements

• (4a) copy annotations for LDT
• (4b) copy annotations for ParamElement

Both steps have the same following functions. They are only different from the referencing mechanism for DTElement and ParamElement.

1. Search the partner annotation document for all parameter elements (or LDT) with the matched hierarchicalName.
2. Copy their annotations to the new annotation document of the client.
3. Update the parameter element (or LDT) in the client annotation document with the new referencing hierarchicalName.

Example for given a match \( \{ \text{HotelBooking, parameters, personName} \} \rightarrow \{ \text{TravelBooking, travelParam, bookPerson.name} \} \)

<table>
<thead>
<tr>
<th>WSname</th>
<th>paramSet</th>
<th>message's part</th>
<th>$\text{XPath}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>find:</td>
<td>HotelBooking</td>
<td>.input.parameters.personName</td>
<td></td>
</tr>
<tr>
<td>replace:</td>
<td>TravelBooking</td>
<td>.input.travelParam.bookPerson.name</td>
<td></td>
</tr>
</tbody>
</table>

Figure 47: Referencing mechanism for ParamElement.

The copy process searches the annotation document for all parameter elements with the hierarchicalName starting with:

\( \text{HotelBooking.input.parameters.personName}.* \)

Since the element personName can be a complex type element and may also have children elements, the copy process has to find all elements starting with the name personName, not just personName only. After the annotations are copied to the client annotation document, they are updated with a new hierarchicalName:

\( \text{TravelBooking.input.travelParam.bookPerson.name}.* \)

Note that, the absolute location path can be null when the data selection specifies a whole part of a message (no specific elements inside). The null path thus prompts the function just to find all parameter elements with the attribute hierarchicalName starting without the location path information.
5.5 Annotation conflicts handler

The current output of the matching algorithm is a list of all matches without the handle for duplicate conflicts within that list. Duplication in the final match causes duplication in the UI annotation generation for the same element of the client. In the following section 5.5.1, the annotation conflict is discussed in details. They are arranged in 3 categories: (1) duplicate activities of the same web service in the BPEL document, (2) conflict annotation elements from within one web service, and (3) conflict annotation elements from different web services.

5.5.1 Solution concepts for conflicts handler

5.5.1.1 Duplicated activities of the same web service

a. Description:

Because the matching algorithm (section 5.3) ignores the structured activities in the BPEL document, the same activity of one web service may be used in different stages of the BPEL process. For example, the reply activity to a client (with the same variable) in a successful process and the one in a compensation handler are nothing but one duplicate activity. The value of the reply variable may be different, but the operation is always the same. The cause of redundant duplicate matching records by the algorithm is that the algorithm does not recognize there are such duplicate activities emerged during runtime from the list of input variables, assignments and operations it is at first given to operate with.

b. Solution:

The duplicate activities are defined as operations (receive, onMessage, onEvent, reply, or invoke) from the same web service with the same input/output variables. They are easy to recognize when reading the BPEL document and needed to be filtered for duplication before they can be used by the matching process.

5.5.1.2 Conflict annotation elements between operations of one WS

a. Description

In this category, when several parameter elements in different operations of one web service are semantically equivalent with only one element of exactly one operation of the composed web service, a conflict occurs. However, if they are equivalent with elements which belong to different operations of the composed web service, there are no conflicts. This kind of conflict happens, for example, when an element of the receive variable from the client is used in different operation of one partner web service.
Figure 48: elements of different operations of one web service are semantically equivalent to one element of the composed web service (right)

b. How to recognize?

<table>
<thead>
<tr>
<th>pattern1</th>
<th>pattern2</th>
</tr>
</thead>
<tbody>
<tr>
<td>match1 {v, p, x} op \rightarrow {v1, p1, x1} op1 partnerLink</td>
<td>match2 {v, p, x} op \rightarrow {v2, p2, x2} op2 partnerLink</td>
</tr>
</tbody>
</table>

composed WS with same op

same WS

This kind of conflict can only be recognized after the final matching list is generated. For 2 given matches from the final matching list, they are considered as “conflict in category 2” when:

- The 2\textsuperscript{nd} patterns of the two matches have the same partnerLink attribute. (Of course they belong to different operations because any equal operations have been already eliminated in category 1)
- The 1\textsuperscript{st} patterns of the two matches have the same operation name attribute and specifies the same element for the annotation generation.

c. Problems

When the 2\textsuperscript{nd} patterns of the two matches select, for example, two elements of the same partner WS \(e_1\) and \(e_2\) for the annotation generation, and each of these two elements has a set of UI annotation elements \(A_1(e_1)\) and \(A_1(e_2)\), which are therefore used to copy to the same set \(A(e)\). A conflict occurs when the two sets \(A_1(e_1)\) and \(A_1(e_2)\) have “the same” annotation element \(a_2\) but with different values.

d. Definition of the same UI annotation elements

It is now important to define, when two UI annotation elements are considered as “the same”.
Figure 49: \( a_2 \) is conflict element between two annotation sets from the same partner WS.

The ServFace project has several UI annotation types for annotating an element, such as feedback label, suggestion, validation... [2]. Two UI annotation elements are now considered as “the same”, when they are have the same type and the same specifications (language or platform):

\[
\forall a \in A(e) : a_1 = a_2 \text{ when } a_1\.type = a_2\.type \text{ and } a_1\.spec = a_2\.spec
\]

**e. Solution**

The new set of the annotation elements \( A(e) \) is built on the union of the two sets \( A_1(e_1) \) and \( A_1(e_2) \) from the same partner web service. When conflicts occur between the two sets, only one annotation element of the two is chosen to present the annotation element in the new set. The statement is formulated as follow:

\[
\forall e \in E, e_1 \in E_1, e_2 \in E_2, a_1 \in A_1(e_1) \text{ and } a_2 \in A_1(e_2) : \\
A(e) = A_1(e_1) \cup A_1(e_2) \\
A(e) = \{ a_1 \cup a_2 | a_1\.type \neq a_2\.type \text{ or } a_1\.spec \neq a_2\.spec \} \\
\cup \{ a_1 \cup a_2 | a_1\.type = a_2\.type \text{ and } a_1\.spec = a_2\.spec \}
\]

5.5.1.3 Conflict annotation elements between different WS

**a. Description**

![Diagram](image)

Figure 50: conflicts from different web services

In this category, several elements from different web services are semantically equivalent with exactly one element of the one composed web service. It happens, for example, when an element of the receive variable is simultaneously used to invoke in different operations of the partner web services. It is now unknown which annotation is taken for annotating the parameter element of the composed web service.
b. How to recognize?

<table>
<thead>
<tr>
<th>pattern1</th>
<th>pattern2</th>
</tr>
</thead>
<tbody>
<tr>
<td>match1</td>
<td>match2</td>
</tr>
<tr>
<td>{v, p, x} op \rightarrow {v1, p1, x1} op1 \textit{partnerLink 1}</td>
<td>{v, p, x} op \rightarrow {v2, p2, x2} op2 \textit{partnerLink 2}</td>
</tr>
<tr>
<td>composed WS with same op</td>
<td>different WS</td>
</tr>
</tbody>
</table>

For 2 given matches from the final matching list, the two matches are considered as “conflict in category 3” when:

- The 2\textsuperscript{nd} patterns of the two matches have different \textit{partnerLink} attribute.
- The 1\textsuperscript{st} patterns of the two matches have the same \textit{operation name} attribute and they select the same element for the annotation generation.

c. Problems

\begin{center}
\begin{tikzpicture}
    
    \node [circle, fill=red!30] (a1) at (0,0) {$a1$};
    \node [circle, fill=red!30] (a2) at (1,1) {$a2$};
    \node [circle, fill=red!30] (a3) at (2,0) {$a3$};

    \node (WS1) at (-1,0) {\textit{WS1, op1, e1}}; 
    \node (WS2) at (1,0) {\textit{WS2, op2, e2}}; 

    \path [->] (WS1) edge (a1) 
                (WS1) edge (a2) 
                (WS2) edge (a2) 
                (WS2) edge (a3);

    \node at (0,-1) {$A(e) = A_1(e_1) \cup A_2(e_2)$};
\end{tikzpicture}
\end{center}

Figure 51: $a_2$ is conflict element between two annotation sets from two different partner WS.

When the 2\textsuperscript{nd} patterns of the two matches select two elements from different partner $e_1$ and $e_2$ for the annotation generation, and each of these two elements has a set of UI annotation elements $A_1(e_1)$ and $A_2(e_2)$, which are used to copy to the same set $A(e)$. A conflict occurs when the two sets $A_1(e_1)$ and $A_2(e_2)$ have the same annotation element $a_2$ but with different values.

The conflict annotation element in this category is much more difficult to handle due to the following situations:

1. Only one service of the two conflict partners is invoked under some known conditions and it is decided in runtime environment. It happens, for example, when a client wants to choose a service from the two partners under some conditions customized by the client himself. The situation hereby is declared as “conflict between annotation elements with different conditions”.

(2) A partner service is always used by default and the other partner services are active only under some known conditions dictated in runtime environment. It happens, for example, when a client wants to use a partner service or not (e.g. book an auto) after he uses the default one (e.g. book a flight). The situation is declared as “conflict between a default annotation element and conditional annotation elements”.

(3) Two partners are active by default or are activated by the same conditions. It is declared as “conflict between the same conditional annotation elements”.

d. Solution

The problem analyses in the previous part shows that the conditions in the BPEL cause troubles but they have not been considered so far. The solution concept comes up with a new definition for default (default AE) and conditional annotation element (conditional AE) in order to resolve these troubles:

Default annotation element: is annotation element which annotates a parameter element. This parameter element belongs to an operation which is always active in runtime (no specific conditions).

∀e ∈ E with E = set ({$v, p, x}, op, partnerLink) and op.cid=0
⇒ ∀a ∈ A(e): a.cid = 0 (default annotation element)
Else: a.cid ≠ 0 (conditional annotation element)

The new annotation for the client is built on the union of all sets of annotation elements from different partners.

∀a1 ∈ A1(e1), a2 ∈ A2(e2):
A(e) = A1(e1) U A2(e2)
A(e) = {a1 v a2 | a1.type ≠ a2.type
or a1.spec ≠ a2.spec } //no conflicts

When conflicts occur between annotation elements, the handle concept for every possible situation of conflict is defined as follows:

(1) For the conflict between annotation elements with different conditions, both conflict annotation elements, together with their conditions, are kept in the new set:
In runtime environment, depending on which runtime conditions are fulfilled, the corresponding annotation element will be chosen.

(2) For the conflict between a default annotation element and conditional annotation elements, all conditional annotation elements (regardless with the same or with different language and platform specifications) are eliminated and only the default annotation element is recorded in the new set:

\[ A(e) = \{ a_1 | a_1.cid = 0 \text{ and } a_2.cid \neq 0, a_1.type = a_2.type \} \]

The rationale behind this is that the default partner service is always active in the BPEL runtime process (the service has no conditions) and the annotation information from the default partner service has a higher priority to be displayed to the client. It is assumed that, he, the client, is interested in the default service much more than the other conditional services, regardless of whether these conditional services are active in runtime environment or not.

Note: although the default service does not contain annotations for a specific specification, which the conditional service has, conditional annotation elements are totally eliminated by the default one. This is for a consistent display.

(3) For the conflict between the same conditional annotation elements (or between default annotation elements), one annotation element from the two conflicts is kept in the new set, but not both of them:

\[ A(e) = \{ a_1 \text{ or } a_2 | a_1.cid = a_2.cid, a_1.type = a_2.type \text{ and } a_1.spec = a_2.spec \} \]

In this case, both annotation elements are not in the new set at the same time because they are all the same. They are defined from the same type and have the same conditions to be activated despite the fact that they may have different values. The reasons of doing this are: firstly, there are no attributes to distinguish both annotation elements in the annotation document, and secondly, they have the same priority to be displayed to the client. Nonetheless, to make a better choice, we may want to pick whichever annotation element that comes from a setting of higher priority.

\[ A(e) = \{ a_1 | \text{ partnerLink}_1.prio > \text{ partnerLink}_2.prio, a_1.cid = a_2.cid, a_1.type = a_2.type \} \]

This priority setting is, in fact, the setting commanded by the administrator, as he prefers to choose the UI annotation from a specific service.
In this section solution concepts are developed to solve the conflicts between annotation elements. The end-result is a new set of annotation elements for different operation elements of the composed WS with the following rules of existence. For one operation element:

**Rule No.1:** conditional annotation elements of the same type from different partners can exist concurrently. But only one of them is chosen to be displayed to the client under known runtime conditions.

**Rule No 2:** when there is a default annotation element, no other conditional annotation elements of the same type are allowed to exist.

**Rule No.3:** under known conditions, there is only one annotation element of a type and with a specific specification exists. That means no annotation elements of the same type with the same specification and conditions are allowed to exist.

### 5.5.2 Extend the UI generation with conflicts handler

Previously a solution concept for handling annotation conflicts is developed. Until now, the current matching algorithm (section 5.3) and the UI annotations generation (section 5.4) are not designed to handle conflicts. The following parts will describe how to extend them with the new features.

#### 5.5.2.1 Matching algorithm & condition ID extension

The conflicts handler solves the conflicts between AEs from different partners by inserting in them with additional conditions (conditional AE). Depending on runtime information, the AE with the satisfied condition is chosen to display to the client. But what are these conditions actually?

In the BPEL, an activity may be under a branch of activities. Each branch is started with a condition element (if-else), presenting various paths the BPEL process may undertake in runtime. The activities (read, write and copy activity) in the chosen path will affect the data flows and result in different matches (client-partner relations) in the end. In other words, a match is now associated with a set of condition IDs, which altogether represent the path that the BPEL process will undertake in runtime in order to result this match.

The conflicts handler requires every match in the final matching list to include with a set of condition IDs, if there are any. Until now the matching algorithm ignores the structured activities. When a match is infused with conditional information, the matching algorithm is required to remain unchanged as much as possible, so that it is still transparent for all BPEL’s structured activities. One simple solution is:

1. When parsing the BPEL document, if a condition element is found (if, else, elif, else), it is set with a unique cid (condition id).
2. Each activity found under the enclosing condition element is set with the same cid, just to mark which condition element the activity belongs to.

\[
\begin{align*}
\text{if (cid1)} & \quad \bullet \text{activity.cid} \leftarrow \text{cid1} \\
\text{if (cid2)} & \quad \text{else if (cid3)} \\
\bullet \text{activity.cid} & \leftarrow \text{cid2} \\
& \text{...} \\
\bullet \text{activity.cid} & \leftarrow \text{cid3} \\
& \text{...}
\end{align*}
\]

Figure 53: Every conditional element & its enclosing activities have the same “cid”

3. The matching algorithm treats every activity as an independent node and runs normally, only with a small modification: when the process encounters one “relevant node” with a cid (not null), it will collect this cid and inserts it in the “temporal match”. In the end every final match will have a collection of cids that the matching process collects on the way.

\[
\begin{align*}
\text{temp2.cid} &= \{1\} \\
\text{temp3.cid} &= \{1, 2\} \\
\text{final match} &= \text{temp3}
\end{align*}
\]

Figure 54: the process adds every CID it finds in the relevant nodes to the temp matching

The figure right above is an example. The matching process collects 2 cids \{1, 2\} on the way and includes them in the final match. The cids of a match presents the path the BPEL process will undertake in runtime that will result in this match. In figure 53, the set \{1, 2\} presents the path on the left, whereby the path on the right may result in another match.

The cids of a match is important for the conflicts handler. If a match has cids, the AEs generated from this match are considered as conditional AEs, if not, they are default AEs.

5.5.2.2 UI annotations generation & conflicts handler extension

The matching process generates a list of all matches. Each match is included with conditional information if there are any. The UI annotations generation will copy all AEs from the partner to the composed WS. This section describes how to handle annotation conflicts during the copy process.

Back to the generation of UI annotations (section 5.4), it has the following steps:
(1) search the partner’s annotation document for a specific set of annotations \(A_0(e)\)
and then (2) copy them to the annotation document of the composed WS. The following part extends the step (2) with the conflicts handler to resolve the conflict between the new set \( A_i(e) \) and an existing set \( A(e) \) in the annotation document of the composed WS, when they both annotate for the same element \( e \).

For every new annotation element in \( A_i(e) \) that the process copies to the new set \( A(e) \) of the composed WS, conflict situations must be scanned for and handled to satisfy the three rules of existence for annotation element.

\[
\begin{array}{c|c}
\text{oldAE} & \text{newAE} \\
\hline
\text{WS1, type1, cid1} & \text{WS2, type2, cid} \\
\hline
\text{WS1, type2, cid2} \\
\hline
\text{WS2, type1, cid3} \\
\end{array}
\]

pool of annotation elements for referenced element \( e \)

**Figure 55:** A new AE from a set \( A_2(e) \) of the WS2 is putting in the pool

Figure 55 illustrates the existing annotation set \( A(e) \) is a pool on the left with several annotation elements (oldAE) and the newAE is a new annotation element is inserted into the pool. Before inserted to the pool, the newAE is checked, if it does conflict with any AE in the pool or not. If it conflicts with an existing AE in the pool, the process has to check if the newAE is a new default AE or a new conditional AE. Why?

1. **Handle for a not conflict AE:**

   If the newAE does not conflict with any AE in the pool, it is inserted to the pool without any problems.

2. **Handle for a new default AE**

   If the newAE has no conditions and there are no default AE (with the same language and platform specification) in the pool, the newAE is considered as a new default AE and is inserted to the pool. Otherwise, the newAE is not inserted to the pool; there are no needs to replace an existing default AE in the pool.

   If the newAE is considered as a new default AE, all existing conditional AEs with the same type in the pool must be removed (according to rule No.1 when there is a default AE, there are no other conditional AEs of the same type are allowed).

3. **Handle for a new conditional AE**
The newAE is considered as a new conditional AE when the following rules are not broken:

1. The newAE is a conditional AE.
2. No default AEs are already in the pool (according to rule No.2).
3. There are no other conditional AEs in the pool with the same type, same specifications and conditions (according to rule No.3 no annotation elements of the same type with the same conditions co-exist). If such a conditional AE exists in the pool, and the newAE is from a different WS with a higher priority setting, it will replace the oldAE.
4. There may be an existing conditional AE from the same type in the pool. But the newAE and the existing conditional AE come from different services.

If the newAE is considered as a new conditional AE, it is added to the pool. Otherwise, there may be a default AE or a conditional AE from the same service which is already in the pool and there are no needs to insert the new one.

All the above handles are now formulated as follows:

<table>
<thead>
<tr>
<th>isConflict</th>
<th>true</th>
</tr>
</thead>
<tbody>
<tr>
<td>newAE.cid = 0?</td>
<td>false</td>
</tr>
</tbody>
</table>

YES:  
isNewDefaultElement ← true  
isNewConditionElement ← false

NO:  
isNewDefaultElement ← false  
isNewConditionElement ← true

For every annotation element in list as oldAE:

If newAE.type = oldAE.type:  
isConflict ← true

If newAE and oldAE from the same WS:

<table>
<thead>
<tr>
<th>Same WS:</th>
<th>Different WS:</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ isNewConditionElement = false</td>
<td>+ if newAE.cid = oldAE.cid:</td>
</tr>
<tr>
<td>+ if newAE.cid = oldAE.cid = 0:</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

If isConflict = false or isNewDefaultElement = true or isNewConditionElement = true: add newAE in list.
5.6 BPEL modification for conditional AE

5.6.1 Modification overview

In the previous section a conflicts handler is developed to decide which AE from the partner service is chosen for annotating a parameter element of the composed WS. Normally, conflict resolutions are based on a default AE. Conditional AEs are used only when there are no defaults. And depending on which path (condition) the BPEL process takes in runtime, a conditional AE with the satisfied conditions is chosen to display in the client GUI.

![Diagram](image1.jpg)

**Figure 56**: normal response (on the left) and response with embedded condition cid (right)

In the figure above, the picture on the left illustrates that the BPEL engine is like a black box in runtime environment and the client knows nothing about what happened inside the BPEL engine. Such information about internal variable instances and conditional behaviors are invisible. For this reason, it is now necessary to modify the BPEL specification so that the BPEL engine will return additional information about the runtime conditional behaviors inside the BPEL engine together with other normal result.

![Diagram](image2.jpg)

**Figure 57**: an “assertion” is inserted right under the branch of interest

To choose the right AE to display to the client, the conflicts handler is interested only in one or more particular paths that the BPEL process may take in runtime. For this purpose, an “assertion” is inserted under the branch of interest so that when the BPEL process access this branch in runtime, the assertion value is set to true (otherwise it remains false to the end of the process) and is returned back to the client in the output variable.
5.6.2 Requirements analysis
Assuming there is a collection of assertions to return back to the client. The assertions are collected within a complex element conditions. The question is where to embed this complex element in the output variable. There are 3 possible locations for element conditions in the output variable: (1) the element is a child element inside the current message’s part of the output variable, (2) or it is copied to a brand new message’s part of the output variable, (3) or it is embedded in the SOAP header of the output variable. The following sub-sections will discuss the limitations of every method and finally conclude that the location (2), a part of a new message, is most suitable for embedding the element conditions in the output variable.

a. Location 1: a child inside the current part

![Diagram of Location 1](image)

**Figure 58: Location 1, new element is a child element inside the current part**

**Description:** the new element conditions is a child element inside an existing message’s part of the WSDL message (figure 58).

**Problem:** this method requires inserting a new element inside an existing part of a message. But it only works when this part is defined from a XSD complex element, neither simpleType nor simple element. SimpleType and simple element can have only text content, but no children elements (example below), so it is impossible to insert a new child element in the content.

```xml
//a new child element has to be inserted inside an existing
//message’s part of the message, but the message’s part is defined
//from a XSD simpleType.
<message>
  <part name="part" type="xsd:string"/> ← cannot insert new child!
</message>
```

**Possible but bad solution:** one possible solution is to redefine the message’s part from a simple to a new complex element. And the new one contains the old simple element under it. But this solution will cause significant changes in the
BPEL as a result, as the location of the old element has changed and every relevant assign activity in the BPEL has to relocate it again.

Moreover, even though the message's part is originally defined from complexType, but this complexType has a choice indicator. Since choice indicator allows only one child element from alternative children elements for the content, it is not suitable to include both two elements: one for the normal result and one for the condition element to return back to the client.

In summary, location 1 for the condition element is not suitable for all general cases.

**b. Location 2: a new part**

![Diagram](image)

**Figure 59: Location 2, new element is mapped to a new part**

**Description**: a new part is defined and is inserted in the WSDL message. The element conditions will be copied to this new part (figure 59).

**Problem 1**: problems occur due to various binding protocols of a web service. The WSDL specification has two binding protocols: **RPC** and **document** (3.3) [4]. According to WS-I compliance ¹ (5.3.1) [36], it is possible for the protocol **RPC** to have multiple parts under one WSDL message and each part has to be defined from a XSD data type. But the protocol **document** only allows one part under one WSDL message and that part has to be defined from a XSD element. Since the BPEL 2.0 specifies under (3.0) [3] that it follows the WS-I Basic Profile [36], inserting a new part inside a message is not allowed in the protocol **document** and the BPEL engine will throw out runtime failure.

**Solution**: to insert more than one part inside a message, the protocol must be converted from document to RPC. It can be done by just redefining all parts in

---

¹ The various Web services specifications are sometimes inconsistent and unclear. The WS-I organization was formed to clear up the issues with the specs. It has defined a number of profiles which dictate how Web services to be interoperable [36].
every message from XSD element to XSD data type. Although this modification of WSDL is possible, changing the binding style of the WSDL forces the client to change his implementation codes when parsing the response message.

**Problem 2:** it is important to remember that the BPEL input/output variable in an operation must be a messageType, or a XSD element which is under the condition that the WSDL message of the operation has only one part defined from this element (10.4) [3].

So if the BPEL’s output variable is a messageType variable, there are no difficulties inserting a new part to the message. But if the output variable is defined from an XSD element, it is not allowed to insert a new part to the message, because the condition of one part for element typed variable will break.

**Solution:** the BPEL designer rarely defines the output variables from element type (normally messageType). However, since the BPEL specification allows this kind of definition, it has to be considered carefully. To solve the problem, the type of the output variable can be modified from an element type to a messageType with some small modifications:

1. A new output variable is defined from messageType. It references to the output message of the reply operation. The new variable is used to replace the old one as one to be replied to the client.
2. The old output variable (element type) is now mapped to the part inside the new output variable.

In summary, location 2 for the element conditions is better than the 1 st one. But it still faces some troubles while depending on the current binding protocol or while the output variable is defined from a XSD element.

c. **Location 3: SOAP header**

**Advantages:** in this method, the element conditions is embedded to the SOAP header of the output message. It has the advantage of being independent from the binding protocol. The binding protocol of the SOAP body keeps unchanged for whatever it is. And inserting a new element in a SOAP header will not change other elements in the header either.

Another upside of this method is that it doesn’t clutter the external interface of a service with information that is not related to its business purpose at all. That means the conditions information for displaying the GUI is completely separated from application data.

**Disadvantages:** although the WSDL specification allows the mapping between the soap:header element with a specific part in a message that defines the header type
(3.7) [4], the downside of this method is that there are no standards for the current BPEL 2.0 specification. BPEL engines like Apache ODE or Oracle support this mapping by their own BPEL extensions. But they are all implementation specific.

Summary of requirements: Assertions are inserted under the branches of interest. Their value will be set in runtime and returned back to the client in the output message. For a common solution to all general cases, it is decided that the condition information is embedded in a new part of the output variable. The binding protocol is always RPC. The modification steps will be described as follows.

5.6.3 Modification steps

a. WSDL modification

(1) Define a new complexType conditionsType:

Assuming $cids$ is a collection of all conditions which are required by the conflicts handler for all conditional AE. Firstly, a new XSD complexType conditionsType is defined. It is a collection of one or more child elements $cid$ from type xsd:boolean (figure 60, on the right).

![Figure 60: WSDL modification](image)

Each child element $cid$ will be used to save the information of an assertion and this assertion is inserted under a branch of interest (the branch with the same $cid$ in the BPEL). So that when returned back to the client, the value of each element $cid$ (true or false), will determine whether a specific branch will be accessed by the BPEL process during runtime or not.

(2) Define a new part

A new part is defined from the above complexType under the WSDL output message (figure 60, on the left). Because of the RPC protocol, the new part must be defined from a XSD data type.

(3) Convert to RPC protocol

Convert the binding protocol from document to RPC if the current binding protocol is document (implementation details in 6.1.4).
b. BPEL modification

Modification in the BPEL has three steps: (1) initiate new variable conditions, (2) collect condition values and (3) embed the conditions in the output variable.

![BPEL modification steps](image)

1. **Initiate variable conditions**

A new BPEL’s variable is defined from the above WSDL conditionsType. It is initiated so that all child elements cid are set to false at the beginning.

2. **Collect conditions values**

Each branch of interest is inserted with an assign activity which sets a child element of variable conditions (the child element with the same cid as the conditional branch cid) to true. The instance value of this element will only be set to true whenever the BPEL process enters this branch in runtime; otherwise it will remain false since the beginning to the end and be returned to the client afterwards.

(!) For this step to work properly, every conditional branch in the BPEL has been already tagged with a cid for identification. This will facilitate the task of finding the correct condition element in the BPEL. Please refer to the previous section 5.5.2.1 for the explanation of cids.

The assign activity must be inserted right after the BPEL condition element but before all other activities. This will avoid the situation whereby other activities are processed before the assign activity in runtime; otherwise the runtime process may either abruptly skip to another sequence of processes or reply to a client immediately without having the assertion set to true.

Following are the right positions for inserting the assign activity under several kinds of condition elements `<if>`, `<elseif>` and `<else>`. 
Using an assign activity, this step copies the instance value of variable `conditions` to the new WSDL part `conditions` of the output variable.

(!) Note that the assign activity must be inserted right before every operation in the BPEL document which replies to the client. But this requirement only applies to the operations that need condition information for handling conflicts, as not all parameter elements are necessarily annotated with conditional AEs. In fact, parameter elements that have only default AEs do not require the conditions information to be included in the reply operation.

5.6.4 Modification output result
The whole concepts in this section describe how to modify the WSDL and BPEL document of the client so that the output message of every operation concerned is embedded with the instance value of variable `conditions`. Note that the variable instance is embedded in the new part `conditions` of the output message. Following is an example of output result:

```xml
<soapenv:Body>
<parameters>  //other part with normal result
  ...
</parameters>
<conditions>  //part conditions
  <cid1>false</cid1>
  <cid2>true</cid2>
</conditions>
```

The part `conditions` contains all necessary information for `conditionalAE` to be displayed in the client GUI. In the example above, it contains 2 parameter elements `cid1` and `cid2`, whereby the branch with the `cid2` is active in runtime. And in this case, the `conditional AE` with the `cid2` is chosen.
5.7 Annotation model extension

With the concept of conditional and default AE, the Annotation model is required to be extended with conditional information, as illustrated in the following figure.

![Diagram showing Annotation Model extension](image)

**Figure 62: Annotation Model extension**

(1) An AnnotationModel can declare a set of global ConditionsSpecification objects. Each of them references to a parameter element `cid` in the part conditions of the output result:

```xml
//global Condition Specification
<conditions hierarchicalName="Service.op.output.conditions.cid" />
```

(2) An annotation object of the ServiceModelElements may contain one or more condition elements. Each of them references to one global ConditionsSpecification object above.

```xml
<referenceObjects hierarchicalName="...">
  <annotations>
    <condition id="/0conditions.0"/> //ref to the 1st global condition spec.
  </annotations>
</referenceObjects>
```

In short, conditional AE is an annotation which references to a set of global condition specifications using the condition elements inside it. Default AE is an annotation which contains no condition elements.
5.8 Concepts realization

Figure 63: Solution concept is realized in 3 parts.

The whole solution concepts are divided into 4 parts: a BPEL Reader, a Matching Generator, an Annotation Generator and a BPEL modifier. Each part has its own requirements and task to complete.

1. **BPEL Reader** realizes the nodes model presentation. It parses the BPEL for all relevant elements and prepares for the use of the Matching Generator. In details the Matching Generator requires a list of all write, read and copy nodes to process. The Matching Generator also requires other information such as: variables, operation, portType, and partnerLink associated with each node activity.

The input of the client WSDL is used for extracting the information of the targetNamespace. With this extracted information, it is decidable which partnerLink in the BPEL corresponds to the client by having the same namespaceURI.

2. **Matching Generator** realizes the matching algorithm to generate all matches between a client and a partner operation in the following presentation form:

\[
(\{v_1, p_1, x_1\}, op_1, plink_1 \rightarrow \{v_2, p_2, x_2\}, op_2, plink_2)
\]

3. **Annotation Generator** generates the annotation document for the client. For this purpose, it receives the matching list from the Matching Generator as the input. The Annotation Generator requires the partner’s annotation document for searching and copying the annotations. During the copy, default and conditional annotation elements are identified.

The partner and client WSDL are used to find missing information for the referencing mechanism of the annotations (see requirements of the annotation generation in 5.4.2.a).

4. **BPEL Modifier** modifies the BPEL and WSDL of the client so that the BPEL engine will return with runtime data, which is used to handle conflicts between conditional annotations. Following the concepts in section 5.6, the modifier requires a list of branches which require inserting assertions, and the information to which operation the assertions’ values should be embedded.
6. IMPLEMENTATION

6.1 Class diagram description

The whole implementation is divided in 4 main classes: a BPEL reader, a Matching Generator, Annotation Generator and BPEL modifier. Their functionalities and requirements are described in the previous section “Concept realization” (section 5.6). The following sections will describe each class in details.

6.1.1 BPELReader

![Class Diagram](image)

**Figure 64: BPELObjects is the container for all relevant BPEL elements.**

With the help of JDOM framework, all relevant elements in the BPEL document are parsed by the BPEL reader and stored in one BPELObjects. BPELObjects is a container for all objects, which are required for all further processes, the Matching Generator and the BPEL modifier.

⚠️ Information about the relevant elements is collected under “Relevant elements overview” in the rudiment chapter (section 2.2.1)
The matching algorithm (concept 5.3.3) requires a list of all write, read and copy nodes to process. Each write or read node is a service operation. They must be arranged for two running types of the matching process: (1) from BPEL input to Partner input and (2) from Partner output to BPEL output. For this reason, all write and read activities are arranged in 4 categories. The arrangement is described in the table below:

<table>
<thead>
<tr>
<th>List</th>
<th>Description</th>
<th>Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>BPEL input</td>
<td>Activities that write data to the BPEL’s variable and have the attribute partnerLink which references to a client</td>
<td>receive, onMessage, onEvent</td>
</tr>
<tr>
<td>Partner input</td>
<td>Activities that read data from the BPEL’s variable and have the attribute partnerLink which references to a partner</td>
<td>invoke with inputVariable, reply</td>
</tr>
<tr>
<td>Partner output</td>
<td>Activities that write data to the BPEL’s variable and have the attribute partnerLink which references to a partner</td>
<td>invoke with outputVariable receive, onMessage, onEvent</td>
</tr>
<tr>
<td>BPEL output</td>
<td>Activities that read data from the BPEL’s variable and have the attribute partnerLink which references to a client</td>
<td>invoke with inputVariable, reply</td>
</tr>
</tbody>
</table>

In the table, an invoke element is presented by 2 nodes at the same time, 1 node for the read (inputVariable) and one node for the write activity (if it has outputVariable).

Each activity element in the table is defined from class Invoke (details in a). The BPELObjects furthermore contains a list of all BPEL’s variables (details in b) and a list of all copy nodes (details in c).

**a. Parsing write and read activities**

In the BPEL, the following elements receive, reply, onEvent, onMessage and invoke perform a write or read activity on the BPEL’s variable. They all have the same attributes (operation, input/output variables, portType and partnerLink). So these elements can be grouped together and defined by one
common class `Invoke`. Class `Invoke` thus presents a write or a read node in the solution concept.

A write or a read node always belongs to a web service. The attribute `portType` of these elements presents this connection, but it is an optional attribute in the BPEL’s specification. For this reason, one `Invoke` object should reference to one `PartnerLink` object (figure 66), which contains further information of a concrete service.

![Figure 66: Each “Invoke” object references to PartnerLink](image)

The attribute `namespaceURI` of a `PartnerLink` object is the most important attribute. In the BPEL’s specification, `namespaceURI` of a `PartnerLink` will match with the `targetNamespace` of the corresponding WSDL. This attribute is important to decide which `PartnerLink` belongs to which WSDL. By calling the `namespaceURI` of the `partnerLink` that an `Invoke` object references to, any further process can identify to which web service an `Invoke` belongs to.

### b. Parsing BPEL’s variables

In the BPEL, the element `variable` defines a BPEL’s variable with a type (element, data or `messageType`) and references to a real name in the corresponding WSDL. The class `Variable` presents this information in attributes `type` and `realName` respectively.

![Figure 67: Class “Variable”](image)

### c. Parsing assign activities

All assign activities collected in the BPEL are parsed to `Assignment` objects. One `Assignment` object presents a copy node. It references exactly to two `Copy` objects “from” and “to”, presenting the data selection FROM and TO of an assign activity.
Furthermore, each Assignment object has an attribute “id”, which is used to identify a node for avoiding the endless run of the matching algorithm (concept 5.3.3).

![Diagram of Assignment and Copy objects](image)

**Figure 68:** One “Assignment” object consists of 2 “Copy” objects from and to.

In the BPEL, the data selection FROM and TO of an assign activity has various forms. In the solution concept, all forms of data selection must be transformed into only one presentation form: `(${variable, part, xPath})`.

The Copy object also has 3 attributes `variable, part` and `xPath`, presenting the 3-tuple presentation form. Further detailed information about the transformation will be described in section 6.2.1.

**d. Parsing condition elements**

In the BPEL, condition elements (if, elseif, else) specify several branches of activities that the BPEL process may take in runtime. Every time the parser finds a condition element, this BPEL element will be set with a new `id` attribute. This `id` is also given to every relevant activity (write, read and copy activity) under its enclosing element (concept 5.5.2.1).

The attribute `id` of the BPEL condition element is used for locating it again by the BPEL modifier, when it wants to insert an assertion right under the branch it interests. Condition element may have a `name` attribute but this attribute is optional and not unique, so it cannot be used to identify the condition element. For this reason a new `id` is required.

### 6.1.2 Matching Generator

The Matching Generator takes the BPELObjects as an input and generates a list of all matches. It follows the process description in section 5.3.4. In the solution concept, a match has the following presentation form:

<table>
<thead>
<tr>
<th>Client</th>
<th>Partner</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>(${variable, part, xPath}), op, plink</code></td>
<td><code>(${variable2, part2, xPath2}), op2, plink2</code></td>
</tr>
</tbody>
</table>

In the implementation, a match is defined from class Matching as follows.
Figure 69: Class “Matching” and its relevant classes

The presentation form of a match specifies which data selection from which operation and service are matched to each other. In the implementation, one Matching object has the same attributes. It contains two attributes from and to (from class Copy) to specify the data selection of a match, and two attributes myInvoke and partnerInvoke (from class Invoke), which reference to the client’s and the partner’s operation that the data selection belongs to.

About the client and the partner operation that a match references to, the matching process has two running types. For each type, myInvoke and partnerInvoke will reference to a different node. It is shown in the table below.

<table>
<thead>
<tr>
<th>Running type</th>
<th>myInvoke</th>
<th>partnerInvoke</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. from BPEL input to Partner input</td>
<td>BPEL input (start node)</td>
<td>Partner input (final node)</td>
</tr>
<tr>
<td>2. from Partner output to BPEL output</td>
<td>BPEL output (final node)</td>
<td>Partner output (start node)</td>
</tr>
</tbody>
</table>

For example, in running type 1, the start node is the client’s operation that writes data to the BPEL, so myInvoke references to the start node. But in running type 2, from Partner output to BPEL output, the client operation myInvoke references to the final node. This is only a little bit difference between the 2 running types.

Note: by referencing to an Invoke object and an Invoke object previously references to a single PartnerLink object, a match presents a relation between 2 specific web services, identified by the PartnerLink object.

6.1.3 Annotation Generator
The task of the Annotation Generator is to create the annotation document for the composed web service based on the matching input from the Matching
Generator. Remembering back to the annotation model; the annotation document contains several referenceObjects elements with a unique hierarchicalName. Each referenceObjects groups a set of AEs for annotating one particular parameter element of an operation.

Figure 70: Class “AnnotationElement”

a. The copy process

In the implementation, the copy process (concept 5.4.3) searches for all relevant referenceObjects from the partner’s annotation document and then clone them. The cloned referenceObjects are then presented in a Map with a pair of key and value. The key is the new hierarchicalName which is modified for referencing to the parameter element of the composed web service. The value is a list of AnnotationElement objects (figure 70), which are defined as follows. First, each AnnotationElement object has an attribute element referencing to an AE of the referenceObjects.

b. The conflicts handler

AnnotationElement object furthermore has a set of condition cid (the cid is copy from the match, if the match has a set of conditions), an annotation type and a partnerTargetNS. The attribute partnerTargetNS saves the namespaceURI of the partner, from which the annotation comes. These three attributes type, cid and partnerTargetNS are necessary for the conflict handler as follows.

When a new referenceObjects is cloned, the conflicts handler checks if there is an existing referenceObjects with the same hierarchicalName already in the Map. If yes, it will add each AnnotationElement of the new referenceObjects to the existing list of the old referenceObjects. Let’s call it a pool.

If there’s an AnnotationElement object having the same attribute type is inserted to the pool, the attribute cid of an AnnotationElement object determines if it is a
default AE (cid = null) or a conditional AE (cid ≠ null). The attribute partnerTargetNS determines if it is the conflict between the same or between different web services. A handling concept for each situation is already developed in section 5.5.2.2.

c. The conflict handler exception

There are AEs which have the same type but it is not enough to consider them as conflict with each other or not. Such annotations from type Feedback (3.3.1.1) [2] may have additional types, such as help, error, or label, to annotate for the same parameter element. And it’s the same for annotation from type VisualProperty (3.3.1.8) [2], it is possible to have additional properties, such as read-only, obscured, to be used at the same time.

So, 2 annotations from type Feedback (or VisualProperty) are conflict, only when their additional attributes are also the same. For this purpose, an AnnotationElement object has an additional attribute typeProperty beside the attribute type. Together they will solve the conflict for these kinds of annotations.

d. Conditions overhead reduction

(1) Duplicate condition reduction

Returning runtime condition back to the client for conflicts handler is an overhead, since it is not the business data of service. The concept of a default AE is developed to reduce the number of conditional AEs as much as possible. And now there is another try from the implementation aspect.

```
  1  2
 / \ /  \\
 3  4
\  / \  /
{2,3} {2,4}
```

**Figure 71: Conditions reduction example**

The above figure illustrates, there are 2 branches of interests (3) and (4) that are required inserting an assertion under them. The conflicts handler may result two conflict AE$_1$ and AE$_2$ with a set of cids {2, 3} and {2, 4} respectively. Instead of saving the same cid {2} in both of them, the implementation can eliminate this duplicate cid and save only the difference between the two sets, resulting only {3} and {4}.

When an AnnotationElement object is inserted to the pool and the conflicts handler recognizes that this is a conflict between 2 AEs from different services
with different set $cid$, and only in this category, the implementation will calculate the different sets of the two and save it in the attribute $diffCid$ of each.

Before the calculation, a security check is performed to ignore the case, when one of the two sets is a sub set element of the other, because in this situation the difference will result an empty set for the sub set $cid$.

$$AE_{new}.diffCid = AE_{new}.cid – AE_{old}.cid \quad (AE_{new}.cid \text{ is not subset of } AE_{old}.cid)$$

$$AE_{old}.diffCid = AE_{old}.diffCid – AE_{new}.cid \quad (AE_{new}.diffCid \text{ is not subset of } AE_{new}.cid)$$

The difference is saved in another attribute $diffCid$, so that the original set $cid$ is kept for recognizing further conflict situations when there are another AnnotationElement object is inserted to the pool.

(2) Default AE for single service reduction

If one parameter element in the final result only has AEs coming from one partner service, these conditional AEs are set to default. The conditions of these AEs are not important, because there are no other conflict partners and the client doesn’t need conditional information for the decision when displaying the GUI of this parameter element.

The implementation is performed after the copy process finished and before the creation of the final annotation document.

**e. Prepare information for the BPEL modifier**

After the copy process finished, the collected referenceObjects in the Map and its AnnotationElement objects are written to the final annotation document.

The BPEL modifier requires a set of cids, which present the branches of interest in the BPEL for inserting assertions. If an AnnotationElement object in the final result has conditions ($cid \neq null$), the conditions are collected in a set for the BPEL modifier. The corresponding operations and messages are also collected in two collections. The operation set contains all operations to embed with runtime condition. The message set contains all messages to define with a new part conditions.

**6.1.4 The BPEL Modifier**

The modifier follows exactly the modification steps in section 5.6.c. It's only necessary to show how the binding protocol is converted from document to RPC. In general, it is a small modification with 3 steps:

(1) In the client WSDL, query the binding element which requires modifying.
   (It is the binding element which binds to the portType containing the operation to embed with the conditions information).
In the following figure, the binding element is modified. The attribute style changes from document to RPC, and the attributes namespace of soap:input and soap:output are added with the targetNamespace of the WSDL.

```xml
<wsdl:binding name="TravelBookingSOAPBinding" type="tns:TravelBook">
  <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/">
    <wsdl:operation name="TravelBooking">
      <wsdl:input>
        <soap:body use="literal" namespace="http://TravelBooking.wsdl" />
      </wsdl:input>
      <wsdl:output>
        <soap:body use="literal" namespace="http://TravelBooking.wsdl" />
      </wsdl:output>
    </wsdl:operation>
  </soap:binding>
</wsdl:binding>
```

Figure 72: Example of binding element with portType “TravelBook”

Furthermore, all parts of operations belonging to the RPC binding portType must reference to a XSD data type, instead of a XSD element. In fact, the referenced XSD data type is the original XSD element without the element name. The figure below is an example:

```xml
<xsd:element name="travelBooking">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element maxOccurs="1" minOccurs="1" name="name of XSD data type"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

Figure 73: Example of XSD data type “travelBooking” from the original XSD element.

With this modification, there are no problems to define a WSDL message with multiple parts.

6.2 Problems and solution
6.2.1 Handling several forms of assign activities
The following part describes how to handle several forms of the BPEL’s assign activities and transform them into the presentation: {$variable, part, xPath}

a. Copy message, message’s part or element

For the data selection of copy activity, the BPEL can select a variable, or a message’s part or an element with the following form:
The attribute `variable` and `part` of the selection element (from/to) are presented by `variable` and `part` in the presentation tuple respectively. The `queryContent` of the query element is an xPath1.0 query to a single XML information item inside the binding variable. It must be transformed into an absolute path inside the binding part (section 6.2.3 below).

There is a different form with the data selection in the text content as follows:

```xml
<from>variable.part/queryContent</from>
```

In this form, the selection is presented in the text content of the selection element. It separates `variable` and `part` with a dot and the `queryContent` with a slash. This form is parsed as the previous one.

b. Copy the whole partnerLink

```xml
//3. Copy the whole partnerLink
<from partnerLink="NCName" endpointReference="myRole|partnerRole"/>
```

This specification can copy the whole partnerLink (or only an endpoint references attribute associated with the partnerLink) to another one. In this case, the transformation has to query all messages in the partnerLink and generate a list of copy nodes. Each copy node presents the copy of one WSDL message between the two partnerLinks in the following presentation form:

```plaintext
{$v, null, null} → {$v, null, null}
```

c. Copy variable’s property

```xml
//BPEL
<from variable="BPELVariableName" property="QName"/>

//WSDL
<vprop:propertyAlias propertyName="QName"
  messageType="QName"? part="QName"?
  <vprop:query>queryContent</vprop:query>
</vprop:propertyAlias>
```

This form specifies the assignment of a variable’s property. The attribute `property` references to an element `propertyAlias` in the corresponding WSDL. Here the transformation has to read the `propertyAlias` in the WSDL for the data selection.

The selection will be transformed into a presentation tuple with `variable` is the `BPELVariableName` of the BPEL’s selection element. `Part` and `XPath` are the
attribute part and the queryContent of the propertyAlias element in the WSDL respectively.

The variable property can be defined as an element in the WSDL (7.3) [3]. In this case the part of the presentation tuple is left as null.

6.2.3 Handling xPath query in BPEL’s variable assignment

The BPEL 2.0 specification can use xPath 1.0 to locate a single XML information item for variable assignment. For the annotation’s referencing mechanism, the xPath query has to be presented in an absolute path presentation. It is the absolute path to the XML information item inside a message’s part or inside an XSD element of the data selection.

\[
\text{WSname paramSet message's part absolute xPath}
\]

\[
\text{TravelBooking.input.travelParam.bookPerson.name}
\]

Figure 74: annotation referencing mechanism needs an absolute xPath

Most BPEL designer tools let the designer choose a child element inside a variable from a drop-down list for the data selection of the copy operation, and then the tool automatically creates an absolute path to the selected element inside the binding variable. But when a designer types his own xPath query, since xPath query has several expression forms, it is unknown if it is an absolute path inside the binding variable or not. And if not, the implementation has to convert it into a full path presentation. In general, the xPath query can be formalized as follows [37]:

\[
\begin{align*}
\text{LocPath} & ::= \text{LocPath}/\text{LocPath} \mid \text{LocPath}[\text{PredExpr}] \mid \text{axis}:\text{node} \\
\text{PredExpr} & ::= /\text{LocPath} \mid /\text{LocPath} \text{VOp const} \\
\text{VOp} & ::= \text{"="} \mid \text{","=} \mid \text{","<"} \mid \text{","<="} \mid \text{",">"} \mid \text{",">="} \\
\text{const} & ::= \text{BooleanExpr} \mid \text{LiteralExpr} \mid \text{NumberExpr} \\
\text{axis} & ::= \text{self} \mid \text{child} \mid \text{parent} \mid \text{descendant} \mid \text{descendant-or-self} \mid \text{ancestor} \mid \text{ancestor-or-self} \mid \text{following} \mid \text{following-sibling} \mid \text{preceding} \mid \text{preceding-sibling} \mid \text{attribute} \\
\text{node} & ::= \text{label} \mid *
\end{align*}
\]

In the location path (LocPath), the predicate expression (PredExpr) is used to select alternative node instances of the same XML element. It can be totally removed, because it doesn’t take any roles in the annotation’s referencing mechanism. Together with the predicate expression, the operation (VOp) and const used by the predicate are also ignored. Now there is only axis:node to consider.
Under 8.2.4 [3], the BPEL specifies that the xPath query cannot access the context node. This restriction of accessing the context node is helpful, because axis nodes such as “parent”, “ancestor” and “preceding” may not be used.

Further works should do researches on the formalization of all xPath query specifications in the future. This protocol doesn’t implement the handles for `axis:node` of xPath query.

6.3 Implementation limitation

The protocol focuses on the implementation of the matching algorithm, the copy annotation process with conflicts handler, and the BPEL modifier. It’s important to prove that their concepts work properly. Although the protocol implements most regular specifications of the BPEL 2.0, there are some restrictions listed as follows.

The protocol neither implements the specifications of using XLST 1.0 function for the copy activity (8.4) [3], nor the use of `axis:node` of xPath query.

Besides, it implements only the current BPEL standardization, but no BPEL engine specific extensions. It has to be announced here, because if an example taken from a web site, such as Oracle or Apache ODE, they should use their own extensions in their examples and the BPEL Reader cannot parse this information and transform them to a nodes model for the Matching Generator. For example, an Oracle’s example may use `switch`, `otherwise` or the function `getVariableData` in their condition elements. These extensions don’t belong to the BPEL 2.0 standard.

The matching algorithm is implemented completely. To extend with new specifications like above, it is only needed to transform new relevant activities to write, read or copy nodes and the matching algorithm will do the rest.

6.4 API

The BPELAnnotator provides the following methods:

- `addPartner (String path1, String path2)`

Add a partner service to the BPELAnnotator with the `path1` and `path2` are the location paths to the annotation document and WSDL of the partner respectively.

- `setClient (String path1, String path2)`

Set the client service to the BPELAnnotator with the `path1` and `path2` are the location paths to the BPEL document and WSDL of the client respectively.

- `run ()`

After setting the client and the partner services, call `run` to start annotating the client service. The client annotation document is created in the same folder of the
client WSDL by default. The modified BPEL is in the same folder of the original BPEL.

- `getMatchingList()`

Return a list of all Matching objects generated by the matching algorithm.

### a. Summary

```java
//init annotator
BPELAnnotator annotator = new BPELAnnotator();
//set client
annotator.setClient(bpelPath, clientWsdlPath)
//add partners
annotator.addPartner(annotationPath1, wsdlPath1);
...
//run
annotator.run();
```

### b. Defaults class

The Defaults class provides some default settings of the tool.

<table>
<thead>
<tr>
<th>Name</th>
<th>Default settings</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>annotationConditionTagName</code></td>
<td>conditions</td>
<td>Global condition element tag name in the client annotation.</td>
</tr>
<tr>
<td><code>wsdlConditionPart</code></td>
<td>conditions</td>
<td>How does the new message’s part condition call?</td>
</tr>
<tr>
<td><code>wsdlConditionDataType</code></td>
<td>conditionsType</td>
<td>How does the new complexType condition call?</td>
</tr>
<tr>
<td><code>bpelConditionVariable</code></td>
<td>conditionVariable</td>
<td>How does the variable for collecting conditions in the BPEL call?</td>
</tr>
<tr>
<td><code>annotationModelURI</code></td>
<td><a href="http://servfaceannotationmodel/1.2">http://servfaceannotationmodel/1.2</a></td>
<td>namespace of the Servface Annotation model.</td>
</tr>
</tbody>
</table>
7. EVALUATION

7.1 Tests

7.1.1 Revisiting the test scenario

The test scenario in the chapter of requirements analysis covers most problems throughout the whole paper in a travel booking service. Received data from the client is used for booking a flight (obligatory service). If the customer wants to book an auto (optional service), there are two auto services to invoke. One of the auto services is invoked if the auto price is cheaper than the other one. Matching table between parameter elements for testing control is at appendix 3.

7.1.1.1. BPEL Reader

The transformation of relevant activities into nodes model has been tested with 30 copy nodes for the internal data flows between variables. The data selection of the copy nodes vary from a message, a message’s part to an element inside the binding variable (appendix 2).

Write and read activities have been tested to operate on the BPEL’s variable from various types (message and element type). While the FlightBS provides the operation travelBooking with variable TravelBookingInput from type element, the variable bookFlightInput, which is used to invoke the operation bookFlight of the FlightBS, is now defined from message type (appendix 1).

7.1.1.2 Matching Generator: test for the chain of data flow

The complexity of the data flow has been tested with nodes that are related to each other in 2 categories: sub set and super set. The following parts will describe 2 examples from the test scenarios.

(1) The 2nd copy node is a super set of the 1st

```
1. destination
2. place
```

bookAutoInput1 bookAutoInput2

parameters

The matching process has the following input nodes:

<table>
<thead>
<tr>
<th>Node</th>
<th>Description</th>
<th>Presentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write</td>
<td>Data received from the client is put in variable TravelBookingInput.</td>
<td>{TravelBookingInput, null, null}</td>
</tr>
</tbody>
</table>
The matching process first starts with the write node. From the write node, it finds the 1st copy node as relevant node, because the data selection FROM of the copy (element destination) is a sub set element of the write node (the whole variable TravelBookingInput). Because this copy node is also related to the 1st read node (variable bookAutoInput1), it is added to the final matching list:

\[
\text{// Expected match 1:} \\
\{\text{TravelBookingInput, null, destination}\} \rightarrow \{\text{bookAutoInput1, parameters, place}\}
\]

From the 1st copy node, the matching process continues to find the 2nd copy node as its relevant node, because the data selection FROM of the 2nd node (the part parameters) is a super set element of the selection TO of the 1st copy node (an element in the part). The temporal matching calculates a new relation:

\[
\text{// Expected match 2:} \\
\{\text{TravelBookingInput, null, destination}\} \rightarrow \{\text{bookAutoInput2, null, place}\}
\]

The calculation depends on the transitive conclusion of category 2 with the super set is an element (table rules in section 5.3.2.2).

Because the new temporal match is related to the 2nd read node (variable bookAutoInput2), it is added to the final matching list as a match. The matching process cannot find any further relevant nodes, it terminates.

(2) The 2nd copy node is a sub set of the 1st
The matching process starts with the write node and finds the 1st copy node relevant, because the selection FROM of the copy node is equivalent to the write node (variable `TravelBookingInput`).

From the 1st copy node, the matching process continues to find the 2nd copy node relevant, because the selection FROM of the 2nd copy node (element `startAirport`) is a sub set element of the selection TO of the 1st (the whole variable `bookFlightInputTemp`). It calculates the new temporal match:

```
// Expected match:
{TravelBookingInput, null, startAirport} → {bookFlightInput, bookFlightRequest, start}
```

The calculation depends on the transitive conclusion of category 3 with the superset `TravelBookingInput` is an element (table rule in section 5.3.2.3). The new temporal match is related to the read node (`bookFlightInput`), it is added to the final matching list.

### 7.1.1.3 Annotation Generator & conflicts handler

#### a. Conflicts from within the same web service

Goal: AEs from within the same web service are merged together. If they are conflict with each other, only one AE is chosen.

<table>
<thead>
<tr>
<th>Node</th>
<th>Description</th>
<th>Presentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write</td>
<td>Data received from the client is put in variable <code>TravelBookingInput</code> (element type)</td>
<td><code>{TravelBookingInput, null, null}</code></td>
</tr>
<tr>
<td>Read</td>
<td>booking a flight with inputVariable <code>bookFlightInput</code> (message type)</td>
<td><code>{bookFlightInput, null, null}</code></td>
</tr>
<tr>
<td>1st Copy</td>
<td>Variable <code>TravelBookingInput</code> is copied to another variable <code>bookFlightInputTemp</code></td>
<td><code>{TravelBookingInput, null, null}</code> → <code>{bookFlightInputTemp, null, null}</code></td>
</tr>
</tbody>
</table>

Figure 75: merge of all AEs from within the same WS

Above is the annotation result for the parameter `startAirport` of the `TravelBS`, whereby AE1 comes from the operation `cancelFlight`. AE2 and AE3 come from the operation `bookFlight` of the same `FlightBS`. These AEs coming from the same web service and they are merged together.
Note that AE₁ and AE₂ have the same type but they have different language specifications (German and English), so they are not conflict and both of them are acceptable in the final result. By modifying AE₁ and AE₂ to have the same language specification, only one of them is copied to the final result. More conflicts from the same web service are documented at appendix 4.1.

b. Conflicts from different web services

Test cases: AEs from different WS are merged together. When conflicts occur, a default AE is chosen. Conditional AEs with different conditions can exist concurrently in the final result. Conditional AEs with the same conditions cannot exist at the same time; except they have different specifications (concept 5.5.1.3).

(1) Conflicts between default AE and conditional AE

The FlightBS (mandatory service) and AutoBS (optional service) have the same label annotation for the parameter destinationAirport of the TravelBS. They are “Flight arrival airport” and the “Place to rent an auto”. Because FlightBS is a default service, its annotation is chosen. The customer is interested in the flight arrival airport more than the place where he rents an auto in his traveling holidays. More conflicts are documented at appendix 4.2.a.

Figure 76: default AE from the FlightBS is chosen

Note: although the AutoBS may have AEs of the same type in many different languages, in Germany and in Italian (figure 77), which the default FlightBS doesn’t have, these AEs from the AutoBS are completely avoided in the final result.

Figure 77: conditional AEs from AutoBS

(2) Conflicts between conditional AEs

The two AutoBS have the same annotations for parameter autoBooksID of the TravelBS. Below is the test result, whereby conditional AEs with different conditions exist concurrently in the final result. AE₁ and AE₂ come from AutoBS1. AE₃ comes from AutoBS2.
Figure 78: conditionalAEs exist concurrently in the final result

Note that two conditionalAEs with the same conditions cannot exist together in the final annotation. The above AE1 and AE2 have the same conditions but with different language specifications so they are an exception.

c. Conflicts handler exception

Annotation Feedback and VisualProperty may have several types and properties at the same time. They are not considered as conflict.

Figure 79: 2 Feedbacks (or 2 Visual Properties) are not conflict.

The test result shows 2 Feedbacks (and 2 Visual Properties) are not conflict for annotating the parameter element travelBegin.

d. Condition overhead reduction

(1) Duplicate condition reduction

Duplicate conditions in conditional AEs are removed. In the test scenario, 2 AutoBS are invoked when the client wants to book an auto (1st condition) and the service with the cheaper price (2nd condition) will be chosen and returned an autoBooksID. The conflicts handler results 2 conditional AEs for autoBooksID. They both have the duplicate condition (the 1st condition) to remove.

Figure 80: duplicate condition is removed
(2) Default AE for a single service

If a parameter element has only AEs coming from one partner, the conditions of these AEs are removed. Conditional AEs are set to default AE.

```
<hierarchicalName="TravelBooking.travelBooking.output...toflightName">
 :type="TextFeedback" text="Flight name" />
<condition id="/"/> delete for defaultAE
```

Figure 81: condition 2 is not important

In the test scenario, the parameter toflightName has only AEs coming from the FlightBS. The AE has condition: the flight is booked successfully or not. But this condition is not important since there are no other conflict partner services, so it is removed from the AE. The AE is now set to default.

7.1.1.4 BPEL modifier

In the implementation, the BPEL modifier changes the BPEL automatically so that the BPEL engine will return the correct information about which branches of interest are processed in runtime. The following test concentrates only in the BPEL modifier, when there is conflict between conditional AEs.

To prove that the BPEL modifier works properly, two FlightBS and one client are implemented and deployed completely. The two partners provide their services to book a flight to the client concurrently. Information of the services is documented in the table below:

<table>
<thead>
<tr>
<th>Client</th>
<th>Original WS</th>
<th>For deployment</th>
<th>Environment</th>
<th>Endpoint</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Partner</th>
<th>For deployment</th>
<th>Environment</th>
<th>Endpoint</th>
</tr>
</thead>
</table>

The original client web service (in folder Test2) has the binding protocol document and returns no runtime information for handling conditional AEs. The conflicts

\[\text{conflicts}\]

\[\text{2 This Test2 is the same as the main scenario, only the FlightBS is implemented completely for testing the BPEL modifier in runtime. Note that the main test scenario will report failure “message-typed variable required” when deploying in ODE because it has element-typed variable in input/output operation. Although it is standardized by the BPEL, Apache ODE 1.3.3 doesn’t support it. So it is not the failure of the implementation. For this reason the test 2 is designed to have only message-typed variables for deploying in ODE.}\]
handler results 2 conditional AEs for the parameter hinflugName of the operation reiseBuchen:

```
hierarchicalName="ReisebuchungService.reiseBuchen.output.hinflugName">
xsi:type="TextFeedBack" text="Easy jet flight name">
<condition id="/@conditions.0" />
<condition id="/@conditions.1" />
```

Figure 82: Conflicts between conditional AEs in parameter element “hinflugName”

The original client service was first modified by the BPEL modifier and the binding protocol was converted to RPC. The modified service was saved and ready for deployment. The test continues with the following steps:

1. **Deployment**

   Copy two deployment files of the partner services (FlugbuchungWS.war & FlugbuchungWS2.war) to the deployment folder of Tomcat (webapps).

   Copy the modified client web service (folder Test2/ReisebuchungWS) to the deployment folder of Apache ODE (webapps/ode/WEB-INF/processes). Then start Tomcat.

2. **Send request**

   Use the “Eclipse Web Service Explorer” coming with Eclipse IDE or any test tools like “soapUI” to send data request to the client endpoint in the table as follows:

   ![ReiseBuchungRequest](image)

   Both services are implemented to have a flight on 2007-05-14. The 1st service has a flight from Paris to Dresden, the 2nd one has a flight from Paris to London.

3. **Respond message**

   After sending the request, the soap:body of the respond message is returned with two parts: one part for the normal result of the flight booking service, one part for
the runtime condition. Depending on the request data of destination airport (Dresden or London), if the input destination airport is London, the element cid1 and cid2 of the condition part is set to false and true respectively, which presents the respond message coming from the 2nd partner, otherwise it’s from the 1st.

```xml
<soapenv:Body>
<dns:reiseBuchenResponse xmlns:dns="http://ReisebuchungsWS/ReiseBuchung">
  <parameters>
    <ReiseBuchungResponse>
      <startFlughafen>Paris</startFlughafen>
      <zielFlughafen>London</zielFlughafen> ← flight to London
      <reiseBeginn>2007-05-14T14:32:00.000Z</reiseBeginn>
      <hinflugName>Air Berlin 017</hinflugName>
      <hinflugBuchungsID>256156208</hinflugBuchungsID>
      ...
    </ReiseBuchungResponse>
  </parameters>
  <conditions> ← part conditions
    <cid1>false</cid1>
    <cid2>true</cid2>
  </conditions>
</dns:reiseBuchenResponse>
</soapenv:Body>
```

Figure 83: Shortcut of respond message. The full message is at appendix 5.

Elements cid1 and cid2 are replied in the correct operation (reiseBuchen), where the global condition elements are annotated:

```xml
<conditions hierarchicalName="..reiseBuchen.output.conditions.cid1" />
<conditions hierarchicalName="..reiseBuchen.output.conditions.cid2" />
```

7.1.2 More tests
The following part continues with some small cases that have not been tested in previously: multiple asynchronous operations.

The 3rd test contains 3 services: book rating and 2 book store services. The client gets rating books and then order book from one of the book store services. It has multiple asynchronous receive operations (onMessage), multiple asynchronous reply operations (invoke) to the client and the partner services. It’s for the matching generator to test with multiple write and read nodes. Besides, the asynchronous operation always has input message (although it is an output to the

---

3 The 3rd test is taken from Oracle example. Description & download available at [http://www.oracle.com/technology/pub/articles/matjaz_bpel2.html](http://www.oracle.com/technology/pub/articles/matjaz_bpel2.html)
client) and the Annotation Generator is tested to find AEs of parameter elements in the input message, but not in the output.

It has not been tested for a match between complex elements, or a match between messages. This test has the match between messages BookResponseMessage of the client operation ClientCallback and BookPurchaseCallback of the book store service. The Annotation Generator has to query all parts under the matching message and copy all annotations in each part to the composed web service. This time the part is defined from local data type and the copy process has to find annotations for the local data type.

The composed web service also has other structured activities such as scope, faultHandlers and eventHandlers at different positions of the BPEL. But they do not affect the result of the matching process.
7.2 Advantages and limitations of the algorithm

7.2.1 Advantages

a. Light weight

The two similar works in chapter 4, [33] and [34] strive to develop handling concepts for all BPEL’s structured activities for the data flow analysis. They aim to simulate all behaviors of the BPEL process statically. The complexity of the implementation also grows huge together with the extension of structured activities. Unlike the two works aforementioned, the matching algorithm developed in this paper focuses only on write, read and copy activities as the relevant activities, discarding the rest as irrelevant. These relevant activities are smartly treated as independent nodes, the relations between which are then identified by the matching algorithm. This algorithm is made independent from the structured activities and runtime data.

![Figure 84: an example of structured activities. Write node A and read node C are relevant.](image)

The above figure shows a snapshot example of structured activities, whereby the BPEL process does not follow the top-down order in runtime, it can jump from scope to scope activities by using the link-semantic specification (11.6.2) [3]. The implementation of the matching algorithm does not have to follow the link-semantic, but it can figure out the data flow from a write node A to a read node C, if the data selections of them are relevant.

In summary, if N is the number of all possible BPEL’s specifications and n is the number of the specifications required by the matching algorithm whilst working on the activities selectively identified as relevant, then n is certainly much smaller than N (n << N). With a much smaller number of specifications, the implementation is undoubtedly light weight and simple.

b. Flexible and scalable

Independence from the BPEL’s structured activities is a big advantage. If the BPEL has any improved or altered versions in the future, the parser only needs to focus on new developments of the relevant activities and builds them into the nodes model, in which each activity is presented by a node with the data selection written in the
3-tuple presentation form. Taking these nodes as the input, the matching process will work normally with no changes. In effect, the design becomes scalable because it is subjected to a much lesser degree of change even as the BPEL’s specifications grow significantly larger.

Furthermore, the whole solution concepts are distinctly divided into 4 parts: a node model presentation, a matching algorithm, copy annotations process with conflicts handler, and the modifier. Each part carries out its own job with a formal input requirement as follows:

<table>
<thead>
<tr>
<th>Parts</th>
<th>Input requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>A node presentation</td>
<td>BPEL document</td>
</tr>
<tr>
<td>Matching algorithm</td>
<td>Nodes model with data selection in form {v, p, x}</td>
</tr>
<tr>
<td>Copy and conflicts handler</td>
<td>Matching list {v, p, x}, op, plink (\rightarrow) {v_1, p_1, x_1}, op_1, plink_1</td>
</tr>
<tr>
<td>Modifier</td>
<td>Set of cid, operation, message to modify</td>
</tr>
</tbody>
</table>

Table of input requirements

The sequential design facilitates the extension of each part in the future. The upgrades of any member part do not affect the other parts, as long as the formal input requirements above are kept unchanged.

c. Fast

The complexity of the matching algorithm depends on the number of relative edges between nodes. Normally one node has two or more relevant nodes and the edge between them is a one-direction edge. In this case the complexity of the algorithm is linear (figure 85, on the left).

\[ O(n) = n \cdot 1 \]

![Figure 85: Complexity of the algorithm. Linear (left) and polynomial in n (right)](image)

When every node is relevant with each other, the edges between nodes become two-direction edges. In this case one node locates far more relevant nodes, and so do its peer nodes; the temporal matching of the algorithm thus takes a longer time to find out all relations between the start and the final nodes. The complexity of the algorithm in this case would be polynomial in \(n^2\).
However, the case that every node is relevant with one another (two-direction edges) will not happen. This is proven below that all relations between write, read and copy nodes in the matching algorithm have only one direction edge.

(1) Edges between write, read and copy nodes have one direction

Inside the BPEL engine, the data flows in one direction: received data from external web service is written to variables. Then the data is transferred from element to element between variables, until it is read by an operation.

The matching algorithm is designed to take the same route by having two running types: (1) from the BPEL input to Partner input and (2) from Partner output to the BPEL output. In each running type, it starts from a write node and finds relevant copy nodes and stops by a relevant read node. The algorithm follows a sequence order: write \( \rightarrow \) copy \( \rightarrow \) read. And thus, one copy node or read node is not necessary to query back for one relevant write or copy node respectively (figure 86 on the left).

![Diagram showing data flow and node relationships](image)

**Figure 86: one direction edge between nodes**

(2) Edges between copy nodes have one direction

The following part examines the relation between 2 copy nodes. Figure 85 on the right examines the relation between two copy nodes and shows that the relation between them has one direction edge. Assume that the 1\(^{st}\) node finds the 2\(^{nd}\) one as relevant, because the data selection TO of the 1\(^{st}\) node and the data selection FROM of the 2\(^{nd}\) node are relevant. This relation results one direction edge from the 1\(^{st}\) node to the 2\(^{nd}\) one. But the 2\(^{nd}\) node will not consider the 1\(^{st}\) one as relevant because the data selection TO of the 2\(^{nd}\) node and the data selection FROM of the 1\(^{st}\) one are not relevant. Their relation thus has only one direction edge.

In summary, after examining all possible edges between nodes, it is concluded that one node in the matching algorithm may have two or more relevant nodes but the edge between two pair nodes is a one-direction edge. The matching algorithm is designed to be linear and hence it is fast. In the implementation, duplicate write or read nodes are also recognized to eliminate unnecessary run.
7.1.2 Limitations

In some conflict situations between annotations from optional services, annotations are recorded with conditions. Two conditional AEs from different partners are concurrently displayed to the client, when their conditions are satisfied in runtime.

a. Overhead?

Runtime conditions for conflicts handler are an overhead, since it is not the business data of the service, especially when there is an overwhelming number of values (assuming there are 20 of them or more) to return back to the client.

To minimize the overhead as much as possible, the following mechanisms are used: the concept of default AE reduces the number of conditional AEs (section 5.5.1.3.d); duplicate conditions among AEs and the unnecessary conditions from a single service are also removed (6.1.3.d). The test scenario is a good example. Two Boolean values over a total of 5 conditions are returned back to the client for handling all conflicts in the end.

b. Conflict handler for input parameter element

A conditional AE is chosen from different partners to display to the users depending on which partner is invoked in runtime. But when no request information have been submitted at the input and no concurrent services have been invoked, it is undecidable which annotation from which partner is chosen for displaying the input fields of the GUI, assuming there are no default services at the beginning. Hence, the conflict handler for annotating input parameter element of a service operation has to choose one of the conflict AEs and eliminate the others. This is not the limitation of the conflict handler. It is only the limitation of missing necessary information for handling conflict.
8. SUMMARY

8.1 Key questions

This paper focuses primarily on the automatic generation of UI annotations for the composed web service. It is done by reusing the UI annotations from all partner services that take part in the composition. The whole paper gives solution concepts to 3 key questions, which are summarized as follows:

**Key question 1: How to generate annotations for the composed web service?**

The generation requires the data dependencies between web services: which data from the client is used to invoke a partner operation and which data from a partner operation is replied back to the client. This paper develops concepts to analyze which parameter elements of one web service are used for another web services in the composition. In such a case, the pairs of parameter elements are called semantically equivalent with each other. When they are equivalent, their UI annotations can be copied and re-used; hence the UI of the composed web service does not have to be designed from scratch.

**Key question 2: How to derive the data dependencies in the BPEL?**

For deriving the data dependencies in the BPEL, this paper introduces concepts of a node model and a matching algorithm.

(1) The node model

Each relevant activity in the data flows is modeled as independent nodes with its data selection. The data selection of them is designed in a 3-tuple presentation to facilitate finding the relationship between nodes in the matching algorithm. The node model is also efficient by having one united presentation for different specifications in selecting data of copy activity in the BPEL.

(2) The matching algorithm

The matching algorithm takes the node model as its input and finds relevant data between them. To overcome problems of the BPEL structured activities and missing runtime information of the static analysis, the algorithm considers all nodes independent from each other and finds relation between them on its own.

Two nodes are relevant when their data selections are relevant in 3 categories: equivalent, sub set and super set. And thanks to the concept of **temporal matching**, the algorithm can overcome the long chain of internal data flows between variables and calculates the final relation between front-end relation, between a client and a partner service operation.
It is proud to re-emphasize the advantages of the algorithm. By having smaller specifications required by the algorithm, the implementation is lightweight. The design also has a fine granularity, which divides the whole process in 4 main parts for an easy upgrade; and hence it is scalable for any further versions of the BPEL specification. And finally the algorithm is fast (section 7.2).

With the above properties, the matching algorithm satisfies the requirements in finding semantic and equivalent data between services in elemental details. In comparison to a similar work from [33], this work does not support handling complex-typed element in a write or a copy activity. And another one from [34] has improvements in the Def-Use analysis, but does not satisfy our own needs. Both of them try to cover the BPEL process and follows the sequence order of all structured activities, and thus their implementation is heavy (chapter 4).

**Key question 3: Which limitations of static analysis are expected?**

The limitation causes by the fact that the BPEL process may take different paths (conditional branches) in runtime environment. And without instance data, static analysis will result all possible writers on one variable and possible UI annotations for one parameter element, but it cannot determine exactly which one of them will be used in runtime.

To overcome this limitation, the work from [34] has an over-approximation on XPath expression. It tries to reduce the number of possible writers on one variable by analyzing the xPath expression in the BPEL; but it is also stated in [29] that it is undecidable whether a given XPath expression is satisfiable or not. This paper solves the problems in another way by inserting an assertion (section 5.6) under the branch of interest that the BPEL process may take in runtime. And when the BPEL process accesses this branch in runtime, the assertion is set to true and its value is returned back to the client. Depending on the instance value of the assertion, one of the conflict annotations is chosen precisely.

But this solution concept opens another problem: the overhead of the assertions, since instance values of assertions are not business data of services. The paper again introduces another concept to determine a default annotation (whenever possible) for reducing a large number of overhead. For example:

1. When a parameter element only has annotations copying from one partner service, these annotations will become default, although the operation of this parameter element is under certain conditions to be active in runtime.

2. The thesis especially develops concepts for handling conflict annotations in different situations: conflict from within one web service and
from different web services (section 5.5). Moreover duplicate conditions in annotations are also recognized and removed.

In summary, this paper introduces all possible concepts to overcome the limitation of static analysis so that the system will have a precise decision for choosing the right annotation with the smallest overhead as possible.

8.2 Recommendation for further development
8.2.1 Motivation
It has been mentioned previously that the instance values of assertions are returned back to the client. Section 5.6.2 introduces several positions for embedding the instance values in the BPEL’s output variable. Among them, the position of a new defined WSDL part is currently recommended because it will work in all general cases. The backward of this method is that the binding WSDL protocol has to be a RPC, but not a literal document. Although the two protocols have a small difference in their presentations of XML message (3.3) [4] and the current thesis supports a transformation from document to RPC, it cannot deny that this embedded position is a protocol dependent solution.

The thesis also discussed about the 3rd possible position where the instance values are embedded to the SOAP header of the output message. It has the advantage of being independent from the binding protocol and moreover it does not clutter the external interface of a service with information that is not related to the business purpose. That means the conditions information for displaying the GUI is completely separated from application data. This embedded position is current not recommended because the current standard version of BPEL 2.0 does not support the data mapping to a SOAP header.

But this position is brought to discuss again because the WSDL specification has already allowed mapping data to a SOAP header (3.7) [4]. Many BPEL engines, such as Apache ODE or Oracle already have their own extensions for the mapping. If the WSDL specification has already supported the mapping, it is sooner or later that the BPEL will come with this additional feature in its next version. And then the instance values of assertions are recommended to embed to the SOAP header of the output message.

8.2.2 Modification for embedding assertion to SOAP header
In short, the WSDL has to define which WSDL part maps to the header of the output message, and the BPEL requires extending with additional extension for specifying which instance data is copied to the header in runtime.
a. Modification in WSDL

Instead of inserting a new WSDL part `conditions` in an existing WSDL output message, the new part is now defined in a brand new WSDL message:

```xml
// new message "respHeader" & part "conditions"
<wsl:message name="respHeader">
  <wsl:part name="conditions" element="tns:conditions" />
</wsl:message>
```

And then map the new part `conditions` to the SOAP header:

```xml
// map WSDL part "conditions" to the SOAP header
<wsl:binding name="ReiseBuchungSOAPBinding">
  <soap:binding style="document"/>
  <wsl:operation name="reiseBuchen">
    <wsl:output>
      <soap:header part="conditions" message="tns:respHeader" use="literal"/>
      <soap:body parts="parameters" use="literal" />
    </wsl:output>
  </wsl:operation>
</wsl:binding>
```

b. Modification in the BPEL

```xml
// embed conditionVariable to the header of the output message
<bpel:assign name="EmbeddedCondition">
  <bipel:copy>
    <bipel:from variable="conditionVariable" />
    <bipel:to header="condition" variable="TravelBookingOutput" />
  </bipel:copy>
</bipel:assign>
```

In step 3 of section 5.6.3.b, instead of copying the conditional instance data to a WDSL part (before the output variable is returned to the client), the copy activity
will copy the instance data to the header of the output variable. The above example uses an extension from the BPEL engine Apache ODE [39]. It extends the copy activity with an additional attribute `header` in the selection TO for targeting the data selection FROM to the header of the output variable.

The Apache ODE extension for mapping instance data to the SOAP header is simple. It is possible that the next version of BPEL may use the same similar specification. And then the instance data of assertions are recommended to embed to this position.

8.3 Discussion on further benefits

It is also interesting to examine the composition of business logics from different web services to figure out the data dependencies between them. The concept developed from this paper can be used by further projects, not just those by the ServFace. In fact, the concept for deriving data dependencies between web services in the composition provides many benefits varying from the semantic web service, web services replacement and compiler optimization.

Semantic web service enables machine-to-machine interaction. The matching algorithm provides benefits for the semantic web service, by specifying which parameter elements from web services are semantically equivalent with each other. These parameter elements have the same semantic meaning and can use the same vocabulary (OWL [21] for example). So the matching algorithm is not only limited in the use of copying UI annotations, but also useful for the interchange of semantic data. When one input/output parameter of a service has its semantic information missing, it can be “copied” by another equivalent one.

This opens to another benefit for programmers to combine data from different sources without losing meaning. Because the web services exist in a dynamic environment, it is possible that a partner in the composition becomes unavailable. And because the service providers are usually bound by a contract with SLA (Service Level Agreement) to provide a certain level of Quality of Service, they may want to keep the service quality by replacing the unavailable partner with another one. The replacement process will be fasten, if the internal data flows inside the BPEL specification are transparent. The administrator does not need to know about the complexity inside the BPEL. The matching algorithm will draw an overview, which input parameter from the client is used under which conditions in a partner’s operation (and which output from a partner returns back to the client). Thus he can bind the interface of the new service to the composition easily.

In the area of compiler optimization, the algorithm is useful to analyze which uninitiated variable (output variable) is defined by initiated variable (input variable); or it checks if a design has failures somewhere in a complex BPEL so that a written variable has never been read (when no matches found).
REFERENCES

2. The ServFace Annotation Model version 1.2. SAP, TUD & Lyria, 10 September 2009.


Dependence Graph. ACM Transactions on Programming Languages and Systems, 13(4), 1991.


38. Apache Tomcat 6.0, the Apache Software Foundation. Available at http://tomcat.apache.org/

39. Apache ODE 1.3.3, the Apache Software Foundation. Available at http://ode.apache.org/
## APPENDIX

1. Table of input/output variables

<table>
<thead>
<tr>
<th>Web service</th>
<th>Input variable</th>
<th>Output variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>TravelBS</td>
<td>TravelBookingInput</td>
<td>TravelBookingOutput</td>
</tr>
<tr>
<td></td>
<td>雕 (element)</td>
<td>雕 (message)</td>
</tr>
<tr>
<td>FlightBS</td>
<td>bookFlightInput</td>
<td>bookFlightOutput</td>
</tr>
<tr>
<td></td>
<td>雕 (message)</td>
<td>雕 (message)</td>
</tr>
<tr>
<td>AutoBS1</td>
<td>bookAutoInput1</td>
<td>bookAutoOutput1</td>
</tr>
<tr>
<td>autoBooking</td>
<td>雕 (message)</td>
<td>雕 (message)</td>
</tr>
<tr>
<td>AutoBS2</td>
<td>bookAutoInput2</td>
<td>bookAutoOutput2</td>
</tr>
<tr>
<td>autoBooking</td>
<td>雕 (element)</td>
<td>雕 (message)</td>
</tr>
</tbody>
</table>

2. Table of copy nodes

<table>
<thead>
<tr>
<th>BPEL variables</th>
<th>Part</th>
<th>xPath</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copy nodes for booking a flight</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TravelBookingInput</td>
<td>null parameters</td>
<td>TravelBookRequest.startAirport</td>
</tr>
<tr>
<td>TravelBookingInput</td>
<td>null parameters</td>
<td>TravelBookResponse.startAirport</td>
</tr>
<tr>
<td>TravelBookingInput</td>
<td>null parameters</td>
<td>TravelBookRequest.destinationAirport</td>
</tr>
<tr>
<td>TravelBookingInput</td>
<td>null parameters</td>
<td>TravelBookResponse.destinationAirport</td>
</tr>
<tr>
<td>TravelBookingInput</td>
<td>null parameters</td>
<td>TravelBookRequest.personNumber</td>
</tr>
<tr>
<td>TravelBookingInput</td>
<td>null parameters</td>
<td>TravelBookResponse.seats</td>
</tr>
<tr>
<td>TravelBookingInput</td>
<td>null null</td>
<td>null</td>
</tr>
<tr>
<td>bookFlightInput</td>
<td>null bookFlightRequest</td>
<td>TravelBookRequest.startAirport start</td>
</tr>
<tr>
<td>bookFlightInput</td>
<td>null bookFlightRequest</td>
<td>TravelBookRequest.destinationAirport</td>
</tr>
<tr>
<td>bookFlightInput</td>
<td>null bookFlightRequest</td>
<td>TravelBookRequest.personNumber</td>
</tr>
<tr>
<td>bookFlightInput</td>
<td>null bookFlightRequest</td>
<td>TravelBookResponse.numberofPersons</td>
</tr>
<tr>
<td>bookFlightInput</td>
<td>null bookFlightRequest</td>
<td>TravelBookRequest.travelBegin</td>
</tr>
<tr>
<td>bookFlightInput</td>
<td>null bookFlightRequest</td>
<td>TravelBookResponse.startDatum</td>
</tr>
<tr>
<td>bookFlightOutput</td>
<td>bookFlightResponse parameters</td>
<td>TravelBookResponse.toflightName</td>
</tr>
<tr>
<td>Copy nodes with cid = 3 (if auto required)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TravelBookingInput</td>
<td>null parameters</td>
<td>TravelBookRequest.destinationAirport place</td>
</tr>
<tr>
<td>TravelBookingInput</td>
<td>null parameters</td>
<td>TravelBookRequest.travelBegin begin</td>
</tr>
<tr>
<td>TravelBookingInput</td>
<td>null parameters</td>
<td>TravelBookRequest.travelEnd end</td>
</tr>
<tr>
<td>bookAutoInput</td>
<td>null parameters</td>
<td>null</td>
</tr>
<tr>
<td>Copy nodes with cid = {3, 4} (If AutoBS1 is chosen)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bookAutoOutput</td>
<td>parameters bookAutoInput2</td>
<td>bookingsResponse.autoID</td>
</tr>
<tr>
<td>TravelBookingOutput</td>
<td>parameters bookAutoInput2</td>
<td>TravelBookResponse.autoID</td>
</tr>
<tr>
<td>bookAutoOutput</td>
<td>parameters bookAutoInput2</td>
<td>bookingsResponse.bookingsID</td>
</tr>
</tbody>
</table>
3. Matching table for testing control
   
a. Received data from client for booking a flight. Output data from FlightBS is replied back to client.

<table>
<thead>
<tr>
<th>TravelBS</th>
<th>FlightBS</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Operation</strong>:</td>
<td><strong>Operation</strong>:</td>
</tr>
<tr>
<td>travelBooking</td>
<td>bookFlight</td>
</tr>
<tr>
<td><strong>Input message</strong>:</td>
<td>bookFlightRequest</td>
</tr>
<tr>
<td>Message’s part:</td>
<td>parameters</td>
</tr>
<tr>
<td><strong>Element</strong></td>
<td><strong>Element</strong></td>
</tr>
<tr>
<td>TravelBookRequest:</td>
<td>BookFlightResponse:</td>
</tr>
<tr>
<td>+ startAirport</td>
<td>+ startFlightairport</td>
</tr>
<tr>
<td>+ destinationAirport</td>
<td>+ destinationFlightairport</td>
</tr>
<tr>
<td>+ personNumber</td>
<td>+ startTime</td>
</tr>
<tr>
<td>+ travelBegin</td>
<td>+ flightName</td>
</tr>
<tr>
<td>+ toflightName</td>
<td>+ seats</td>
</tr>
<tr>
<td>+ seats</td>
<td>+ bookingsID</td>
</tr>
<tr>
<td>+ toflightBooksID</td>
<td></td>
</tr>
<tr>
<td><strong>Output message</strong>:</td>
<td>bookFlightResponse</td>
</tr>
<tr>
<td>Message’s part:</td>
<td>parameters</td>
</tr>
<tr>
<td><strong>Element</strong></td>
<td><strong>Element</strong></td>
</tr>
<tr>
<td>TravelBookResponse:</td>
<td>BookFlightResponse:</td>
</tr>
<tr>
<td>+ startAirport</td>
<td>+ startFlightairport</td>
</tr>
<tr>
<td>+ destinationAirport</td>
<td>+ destinationFlightairport</td>
</tr>
<tr>
<td>+ travelBegin</td>
<td>+ startTime</td>
</tr>
<tr>
<td>+ toflightName</td>
<td>+ flightName</td>
</tr>
<tr>
<td>+ seats</td>
<td>+ bookingsID</td>
</tr>
<tr>
<td>+ toflightBooksID</td>
<td></td>
</tr>
</tbody>
</table>

b. Received data from client for querying auto price

<table>
<thead>
<tr>
<th>TravelBS</th>
<th>AutoBS</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Operation</strong>:</td>
<td><strong>Operation</strong>:</td>
</tr>
<tr>
<td>travelBooking</td>
<td>priceQuery</td>
</tr>
<tr>
<td><strong>Input message</strong>:</td>
<td>autoBookingRequest</td>
</tr>
<tr>
<td>Message’s part:</td>
<td>parameters</td>
</tr>
<tr>
<td><strong>Element</strong></td>
<td><strong>Element</strong></td>
</tr>
<tr>
<td>TravelBookRequest:</td>
<td>BookPriceResponse:</td>
</tr>
<tr>
<td>+ destinationAirport</td>
<td>+ place</td>
</tr>
<tr>
<td>+ travelBegin</td>
<td>+ begin</td>
</tr>
<tr>
<td>+ travelEnd</td>
<td>+ end</td>
</tr>
<tr>
<td><strong>BPEL’s variable</strong>:</td>
<td>autoPrice</td>
</tr>
<tr>
<td><strong>Output message</strong>:</td>
<td>bookPriceResponse</td>
</tr>
<tr>
<td>Message’s part:</td>
<td>parameters</td>
</tr>
</tbody>
</table>
c. Received data from client for booking an Auto. Output data from an AutoBS is replied back to the client.

<table>
<thead>
<tr>
<th>TravelBS</th>
<th>AutoBS</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Operation:</strong> travelBooking</td>
<td><strong>Operation:</strong> autoBooking</td>
</tr>
<tr>
<td><strong>Input message:</strong> travelBookingRequest</td>
<td><strong>Input message:</strong> autoBookingRequest</td>
</tr>
<tr>
<td><strong>Message’s part:</strong> parameters</td>
<td><strong>Message’s part:</strong> parameters</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Element</th>
<th>No parent element</th>
</tr>
</thead>
<tbody>
<tr>
<td>TravelBookRequest:</td>
<td></td>
</tr>
<tr>
<td>+ destinationAirport</td>
<td>+ place</td>
</tr>
<tr>
<td>+ travelBegin</td>
<td>+ begin</td>
</tr>
<tr>
<td>+ travelEnd</td>
<td>+ end</td>
</tr>
</tbody>
</table>

| Output message: travelBookingResponse | Output message: autoBookingResponse |
| **Message’s part:** parameters | **Message’s part:** parameters |

<table>
<thead>
<tr>
<th>Element</th>
<th>Element bookingsResponse:</th>
</tr>
</thead>
<tbody>
<tr>
<td>TravelBookResponse:</td>
<td></td>
</tr>
<tr>
<td>+ autoID</td>
<td>+ autoID</td>
</tr>
<tr>
<td>+ autoBooksID</td>
<td>+ bookingsID</td>
</tr>
</tbody>
</table>

4. Table of conflict parameter elements

4.1 Conflicts from within the same WS

a. Conflict elements from within FlightBS

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input message:</strong> bookFlightRequest</td>
<td><strong>Input message:</strong> cancelFlightRequest</td>
</tr>
<tr>
<td>+ start</td>
<td>+ startFlightairport</td>
</tr>
<tr>
<td>+ destination</td>
<td>+ destinationFlightairport</td>
</tr>
</tbody>
</table>

b. Conflict elements from within AutoBS

<table>
<thead>
<tr>
<th>Operation: priceQuery</th>
<th>Operation: autoBooking</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input message:</strong> autoBookingRequest</td>
<td><strong>Input message:</strong> autoBookingRequest</td>
</tr>
<tr>
<td>+ place</td>
<td>+ place</td>
</tr>
<tr>
<td>+ begin</td>
<td>+ begin</td>
</tr>
<tr>
<td>+ end</td>
<td>+ end</td>
</tr>
</tbody>
</table>

4.2 Conflicts from different WS

a. Conflict elements between FlightBS and AutoBS

<table>
<thead>
<tr>
<th>FlightBS</th>
<th>AutoBS</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Operation:</strong> bookFlight</td>
<td><strong>Operation:</strong> autoBooking</td>
</tr>
<tr>
<td><strong>Input message:</strong> bookFlightRequest</td>
<td><strong>Input message:</strong> autoBookingRequest</td>
</tr>
<tr>
<td>+ destination (destination airport)</td>
<td>+ place (the place to rent the auto)</td>
</tr>
<tr>
<td>+ startDatum (Travel begin time)</td>
<td>+ begin (start rent time)</td>
</tr>
</tbody>
</table>

b. Conflict elements between AutoBS1 and AutoBS2

<table>
<thead>
<tr>
<th>AutoBS1</th>
<th>AutoBS2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Operation:</strong> autoBooking, priceQuery</td>
<td><strong>Operation:</strong> autoBooking, priceQuery</td>
</tr>
</tbody>
</table>
### 5. Test 2 - Respond message

```xml
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:odens="http://ReisebuchungsWS/ReiseBuchung">
  <soapenv:Body>
      <ReiseSuchungResponse>
        <buchungOK>true</buchungOK>
        <startFlughafen>Paris</startFlughafen>
        <zielFlughafen>London</zielFlughafen>
        <reiseBeginn>2007-05-14T14:32:00.000Z</reiseBeginn>
        <reiseEnde>2000-01-01T11:00</reiseEnde>
        <hinflugName>Air Berlin 017</hinflugName>
        <rueckflugName/>
        <plaezte>3</plaezte>
        <hinflugBuchungsID>250156208</hinflugBuchungsID>
        <rueckflugBuchungsID>0</rueckflugBuchungsID>
        <autoBuchungsID>0</autoBuchungsID>
        <autoID>0</autoID>
        <hotelBuchungsID>0</hotelBuchungsID>
        <hotelID>0</hotelID>
      </ReiseSuchungResponse>
    </odens:reiseBuchenResponse>
  </soapenv:Body>
</soapenv:Envelope>
```