An Interface Approach to Discovery and Composition of Web Services

by

Shahram Esmaeilsabzali

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2004

©Shahram Esmaeilsabzali 2004
I hereby declare that I am the sole author of this thesis.
I authorize the University of Waterloo to lend this thesis to other institutions or individuals for the purpose of scholarly research.

Shahram Esmaeilsabzlali

I further authorize the University of Waterloo to reproduce this thesis by photocopying or other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Shahram Esmaeilsabzlali
The University of Waterloo requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.
**Abstract**

As software artifacts in more abstract forms, such as components and services, are becoming more prevalent, formal methods are required that are capable of reasoning about their behaviors, similarities and composition. We introduce a set of formalisms for modelling such systems. Our models are based on interface models and as such adhere to their well-formedness criteria. We define the necessary operations on our models and prove their well-formedness. We also introduce satisfiability relations which correlate different types of interfaces.

As a case study, we study Web services and show how our proposed formalisms can be applied for modelling Web services. Furthermore, we provide methods for discovery and composition of Web services.
Acknowledgements

I am grateful to have had Prof. Farhad Mavaddat as my supervisor. This work has not been possible without his support. I have learned a lot working with him. His enthusiasm for excellence has been a constant motivation for me to move forward. I also enjoyed his constructive skepticism that taught me to rethink the way I deal with science.

Seyyed Vahid Hashemian has been a great friend and assistance in this work. He has made significant contributions to this thesis. My discussions with him have always been a source for inspiration and new ideas.

I would also like to thank the readers of this thesis, Prof. Nancy Day and Prof. Paulo Alencar, for their comments and insights that have made this thesis a better work.

Mahdi Mirzazadeh has also been of great support and assistance by his friendship and vast knowledge in Algorithms.

I thank Evelyn and Don Frizzell for their kindness. I felt at home living in their house and was able to concentrate on my work in a perfect atmosphere.

I thank my parents and my brothers, Bahram, Hadi and Kourosh, for their love and support. They have always been remotely a source of energy.
For my parents, Esmat Yari and Abdolkarim Esmaeilsabzali
# Contents

1 Introduction and Motivation  
1.1 Problem Statement .......................... 1  
1.2 Motivation .................................. 1  
1.3 Approach .................................... 3  
1.4 Contributions ................................. 6  
1.5 Outline ...................................... 7  

2 Interface Models  
2.1 Interface Theories .......................... 11  
  2.1.1 Interfaces/Components Formally ........ 14  
  2.1.2 Stateless Input/Output Interface ....... 16  
  2.1.3 Stateless Port-Dependency Interface .. 17  
  2.1.4 Weak Interfaces ......................... 19  
2.2 Stateless Multiple Port Dependency Interface  
  2.2.1 Connection .............................. 22  
  2.2.2 Refinement ............................... 23  
  2.2.3 Composition ............................. 24  
  2.2.4 MPDs are interfaces .................... 26  
2.3 Enhanced Stateless Port Dependency Interface  
  2.3.1 Some Definitions/Notations ............ 30  
  2.3.2 Connection .............................. 34  
  2.3.3 Refinement ............................... 36  
  2.3.4 Composition ............................. 36
6.3.1 Strong Satisfiability Algorithm ........................................ 154
6.4 Web Services Satisfaction .................................................. 159
6.5 Semantic Matching of Web Services ...................................... 161

7 Composing Web Services .................................................... 165
  7.1 Related Works ............................................................. 166
    7.1.1 Petri Net Approach ................................................. 167
    7.1.2 Finite State Machine Approach ................................... 168
    7.1.3 State Charts Approach .............................................. 170
    7.1.4 A Semi-Automatic Approach ...................................... 172
  7.2 Composition Algorithm .................................................. 173
    7.2.1 Decomposition of Requests ....................................... 173
    7.2.2 Preprocessing of the Requests .................................... 175
    7.2.3 Algorithm ............................................................ 178
    7.2.4 Algorithm Description ............................................. 184
  7.3 Glue Code Generation .................................................... 185
  7.4 Advantages of This Approach .......................................... 187

8 Conclusions and Future Work .............................................. 189
  8.1 Summary and Conclusions ............................................... 189
  8.2 Future Work ............................................................. 192

A Proofs ................................................................................. 195
  A.1 Chapter 2 Proofs .......................................................... 195
  A.2 Chapter 3 Proofs .......................................................... 209

B Glossary of Symbols .......................................................... 217
  B.1 Chapter 2 Symbols ....................................................... 217
  B.2 Chapter 3 Symbols ....................................................... 219
List of Tables

4.1 WSDL message exchange patterns ........................................... 98

6.1 Correlation between dependency pairs and WSDL patterns .......... 160
6.2 Different levels of semantic matching between messages ............ 163
List of Figures

2.1 Hierarchy relation on Components and Interfaces. .......................... 13
2.2 IO interfaces ................................................................................. 18
2.3 PD interfaces ................................................................................ 20
2.4 Two Composable MPDs ................................................................. 26
2.5 Two Composable MPDs ................................................................. 27
2.6 Composition of two MPDs .............................................................. 27
2.7 Two Composable EPDs ................................................................. 38
2.8 Two Composable EPDs ................................................................. 39
2.9 EPD Dependency Computation Algorithm ...................................... 43
2.10 Composition of two EPDs ............................................................ 44
3.1 An IA representing a complex Web service .................................... 55
3.2 Sample IA ..................................................................................... 57
3.3 Sample IA ..................................................................................... 58
3.4 Product of two IAs ......................................................................... 59
3.5 Legal environment for product of two IAs ........................................ 60
3.6 Product of an environment with an IA ............................................ 60
3.7 Composition of two IAs ................................................................. 62
3.8 Refinement of an IA ...................................................................... 63
3.9 Composition of two IAs ................................................................. 65
3.10 A PIA ......................................................................................... 67
3.11 Algorithm for removing illegal states from PIA ............................... 76
3.12 Composition of two PIAs ............................................................. 82
3.13 Refinement in PIA ........................................................................ 86
3.14 Composition of two PIAs (Refinement of Figure 3.12) ................................... 87

4.1 Schematic of Web Service Execution ............................................................... 91
4.2 WSDL Elements .............................................................................................. 94
4.3 A sample WSDL operation. ........................................................................... 96
4.4 UDDI core data structures ............................................................................. 107
4.5 Structure diagram of tModel ......................................................................... 110
4.6 Structure diagram of businessEntity .............................................................. 112
4.7 Structure diagram of businessService ........................................................... 113
4.8 Structure diagram of bindingTemplate ......................................................... 114
4.9 Stack of Web services technologies ............................................................... 118
4.10 A PIA in loop. ............................................................................................... 124

5.1 OWL-S Hierarchy ........................................................................................... 135
5.2 Mapping atomic process to OWL-S ............................................................... 141
5.3 Mapping of a composite process to PIA ......................................................... 143
5.4 Partial OWL-S Specification. ....................................................................... 144

6.1 Sample EPD for specifying user request. ....................................................... 150
6.2 Satisfiability relation illustrated .................................................................... 153
6.3 Algorithm for checking the Strong satisfiability between PIA and EPD. ...... 158

7.1 Decomposition of a Request ........................................................................... 176
7.2 Composition of EPDs. ................................................................................... 177
7.3 Composition algorithm .................................................................................. 183
7.4 Environment automaton for glue code creation ........................................... 186
Chapter 1

Introduction and Motivation

1.1 Problem Statement

The purpose of this thesis is to provide formalisms, methods and algorithms for modelling software components and services and reasoning about similarities among instances of such models and their composition.

1.2 Motivation

Software artifacts in forms of components and services are inherently meant to be reusable. Their specification can be published and advertised in registries and can be discovered by parties that are interested in their functionalities. There are different challenges for such a component/service driven paradigm:

- How such a public registry, for components/services, can be created?
- How specifications of components/services can be stored in registries?
- How search requests against the registry can be specified?
- How search requests are compared against components/services specifications?
• How functionalities of multiple components/services can be combined to satisfy a certain search request?

To address such issues, it is important to realize the required level of detail that is appropriate for specifying software artifacts and search requests. Software artifacts are traditionally documented in two respects, namely, specification of their *signatures*, i.e. input and output behaviors and necessary type definitions; and specification of their *behaviors*, i.e. pre-conditions and post-conditions.\(^1\) Such a separation is also helpful in the context of components and services. However, components and services are complex software artifacts and as such require appropriate formalisms to model their characteristics. Similarly, search requests for components and services should be specified by enhanced formalisms that are expressive enough to model the requests in an appropriate level of abstraction, necessary for performing the search.

Considering the problem of specifying the *signature* of components/services, the challenge is: what level of abstraction is appropriate for such specifications? As an example, consider a component that provides a set of functionality through its functions. Furthermore, this component also requires to call some *external* functions to fulfill its tasks. There are different options to model this; we can model the component’s functions as *inputs* and the external function calls as *outputs*. Alternatively, we can model the signatures of functions too. Moreover, we can specify the functions that call each *external* function. It is possible to specify a component’s behavior in different levels of abstraction. Different applications require different levels of abstraction for specifying behavior of a certain system. Similarly, a search request can also be specified in different levels of abstraction.

Assuming there are appropriate formalisms for specifying behaviors of software artifacts and corresponding search requests, there should also exist correlations between such formalisms. Such correlations determine the level of similarity between the request and software artifacts. Determining correlations and similarity between instances of different formalisms is necessary for any searching mechanism.

While a certain search request could be satisfied by a component/service in a registry, it is possible that there is no single software artifact satisfying that search request. However, there could exist multiple components/services, which can work in collaboration, to satisfy

\(^1\) See [ZW96, ZW97] for such a separation of concerns in the context of function/module matching.
such a request. Two main challenges in this respect are:

- Which software artifacts are potentially helpful for satisfying a search request?
- How can we compose such potentially helpful software artifacts?

An alternative approach is to decompose a search request in such a way that the decomposed sub-requests can be satisfied individually. Of course, this approach is acceptable only if the composition of software artifacts, satisfying the sub-requests, results a software artifact which satisfies the original request. The importance of such an approach lies in the fact that, search for such satisfying sub-requests can be carried out in a parallel manner. Such a decomposition can only be helpful, however, if carried out in such a way that results in satisfiable sub-requests.

A recent trend of technology presents software functionalities as publicly accessible services; Web Services are a set of technologies and specifications aimed at creating a loosely coupled platform where their functionalities can be discovered and invoked in a uniform way over the internet. Involved technologies and specifications are meant to provide a minimal description of the services enabling interested parties to discover and deploy their desired services easily. While different Web service specification standards introduce formats\(^2\) for specifying Web services functionalities, they are not based on any specific formal model that consistently represents Web services as a whole. To achieve a coherent vision of Web services, which provides means for reasoning about their characteristics and carrying out operations on them, as described above, it is essential to model them using appropriate formalisms/models.

### 1.3 Approach

In this thesis, our approach for addressing the problems outlined in the previous section is mainly formal. We present different sets of formalisms, each aiming at modelling a certain aspect of component/service oriented software development paradigm.

---

\(^2\)All such formats are XML-based.
CHAPTER 1. INTRODUCTION AND MOTIVATION

The models we propose are based on interface models [dAH01b]. Interfaces are concise models meant to specify how systems can be used. All interface models follow certain well-formedness criteria. The two main characteristics of interface models are they rely on helpful environments and they support top-down design.

Interfaces assume that there exists a helpful environment which provides their required inputs appropriately. As such, interfaces do not have to deal with fault scenarios, in which, inputs are not provided appropriately. This is a realistic assumption in the context of software systems. Users of software systems are expected to know the necessary inputs and the temporal order of such inputs to achieve a certain goal. The assumption of existence of helpful environments leads to concise and efficient representation of systems.

Top-down design is another important well-formedness criteria for interface models. An interface can be replaced by a refined version. As such, refinement relation is a key relation for any interface model. A more refined interface should be able to work within the existing environment, i.e. with existing set of inputs. It should also provide the output expectations that other interfaces and environment have from the original interface.\(^3\) Other interfaces may rely on a certain interface’s outputs and thus the refined version of an interface should be capable of satisfying such expectations. By composing two interfaces they can work in collaboration. In fact, all interface models are expected to provide refinement and composition operations as part of interface models’ well-formedness criteria.

In this thesis, we introduce three new interface models, each capable of modelling software systems in a different level of abstraction. We start by defining a stateless model, namely, Stateless Multiple Port Dependency Interface, that is capable of specifying the inputs and outputs of systems, as well as, dependencies among those elements. This model is stateless in the sense that it is not capable of capturing the temporal order of execution and events in a system. Instead, it models externally observable behaviors of a system in a high level of abstraction. Such a picture can sometimes be helpful when we are intentionally interested in specifying a certain system in a high level of abstraction.

The second model we propose, Enhanced Stateless Port Dependency Interface, is neither entirely stateless nor entirely stateful. This model is capable of expressing certain stateful characteristics of stateful systems, however, it does not enumerate all possible states of a

\(^3\)Such refinement criteria are manifested differently in different types of interface.
certain system, as stateful interfaces are required to do. Composition of two such interfaces
does not consider all possible interleavings of involving interfaces’ states. Instead, only a
natural scenario of executions for two interfaces are considered valid. Our attempt to
create an intermediary model, between stateless and stateful, resulted in a model that
is not completely compatible with interfaces’ well-formedness criteria. Composition, in
interface models, is a binary commutative and associative operation. In our model, the
composition is not associative and thus results in a model that we call weak interface. It
can be observed that associativity in composition can be achieved only if either there is no
states present or states are fully represented.\footnote{This can be intuitively observed when interface models are described. We do not provide any proof for that though.}

The third model we propose is Parameterized Interface Automata which is a stateful
interface, based on interface automata [dAH01a]. This model is capable of specifying
input/output behaviors of a system as well as the temporal order of events happening in
such a system. Our model, as opposed to interface automata, can specify the parameters
of functions too. We find such level of expressiveness useful.

As a case study, we show how parameterized interface automata are suitable for mod-
elling Web services. Having Web services modelled as parameterized interface automata,
we show how we can formally reason about their characteristics and how we can develop
methods for their discovery and composition. We also show how stateless multiple port
dependency interfaces and enhanced stateless port dependency interfaces are useful in cap-
turing search requests in a Web service discovery system.

For such a Web service discovery system to work, there should exist a relation that
correlates stateless and stateful models. We call such relations satisfiability relations, and
introduce such a relation between enhanced stateless port dependency interfaces and pa-
parameterized interface automata.

While satisfiability relations may not hold between an instance of an interface type and
an instance of another interface type, the composition of some interface instances belonging
to the latter type may be able to satisfy the interface belonging to the former type. In the
context of Web service discovery system, we introduce an algorithm that looks for such
compositions of Web services that can satisfy a certain search request.
1.4 Contributions

The contributions of this thesis are the following:

• **Useful Interface Models:** We introduce three different interface models, each having different expressivity capabilities making them suitable tools for modelling different aspects of software artifacts.

  *Parameterized interface automaton* is a useful formalism for modelling complex component based systems. It is expressive enough to model components’ methods, their parameters and valid temporal orders in which components can operate. We have also defined appropriate operations over parameterized interface automata, i.e. composition and refinement, which enable us to formally reason about characteristics of components. By composing different components and properly integrating their different elements, it is possible to create new enhanced functionalities. Furthermore, by using refinement, it is possible to reason about similarity of two components.

  We also propose two stateless models, *multiple port dependency interface* and *enhanced port dependency interface*, that can be used to specify behaviors of software artifacts in a high level of abstraction. Such levels of abstractions can be useful when we are dealing with systems that their behaviors are not completely known. In this thesis, we propose another interesting application for such models; we use them for modelling search requests in a Web service discovery system.

• **Satisfiability Relations:** We propose satisfiability relations as mechanisms to compare and correlate different types of interfaces. We believe such relations are helpful in bridging between specifications of software artifacts that are presented in different levels of abstraction. Usually, within a formalism, there exists a mechanism for determining similarity between instances of that model. In satisfiability relation, we aim at defining intuitive relations that determine the similarity of models belonging to different formalisms.

• **Web Services Discovery/Composition:** We propose an original approach for discovery and composition of Web services. We provide a coherent set of formalisms and precisely specify different elements of our approach.
Advantages of our approach over other approaches can be categorized as following:

- We use two different sets of formalisms for specifying search requests and modelling Web services. This is crucial since it enables us to capture search requests in an appropriate level of abstraction rather than expecting the requesters to specify all details of their desired Web services, which is an unrealistic assumption. Then, by using satisfiability relations, we are capable of reasoning about similarity of a search request and a certain Web service.

- We propose methods and algorithms for composing Web services such that their composition achieves a certain requested functionality. This is as opposed to most of other approaches where the concept of composition is either under specified or is totally absent. Furthermore, we propose some methods for systematically decompose search requests into some sub-requests that can be more easily checked for satisfaction.

### 1.5 Outline

Chapter 2 describes interface models and their characteristics. Specifically, we focus on describing stateless interface models and present two interface models, namely, Stateless Multiple Port Dependency Interface and Enhanced Stateless Port Dependency Interface. Different operations and characteristics of these two models are specified in details. In Chapter 3, we first describe interface automata and then introduce our new stateful interface model, Parameterized Interface Automata. In Chapter 4, we first provide background information about Web services technologies, which is necessary to understand the rest of the thesis, and then briefly describe workflow technologies and patterns and show their relevance to the Web services technologies. Chapter 5 describes how Web services can be simulated by interface models. We show how a certain semantic Web format, namely, OWL-S, is a suitable format for specification of Web services. In Chapter 6, we introduce the satisfiability relations and present a satisfiability relation which can be applied to the problem of Web service discovery. In Chapter 7, we present a method for composition of Web services. We also propose an algorithm that is capable of looking for helpful Web
services that if composed can satisfy a search request. Also, in the beginning of Chapter 7, we provide a background study on some of the interesting approaches to the problem of Web services discovery and composition. And finally, in Chapter 8, we present our conclusions of this research and lay out the future directions of our studies. Appendix A provides the proofs for Lemmas and Theorems proposed throughout the thesis and Appendix B is a glossary of the symbols and terms used in the thesis.
Chapter 2

Interface Models

*Interface* is a buzz word in the computer field. We often use the term “interface” in different disciplines of computer science and information technology. “Interface” has been used in programming languages, requirement engineering, component-based design, hardware design, Web services (see Section 4.2.1), object oriented techniques and etc, but not always referring to the same notion. In this chapter we provide yet another formal definition of interface and show how we use it in the context of our work.

Luca de Alfaro and Thomas A. Henzinger, in [dAH01b], introduce “interface theories” in the context of component-based design systems. They introduce two types of formalisms: interfaces and components, each formalism aiming at modelling a specific type of systems. In simple terms, interfaces are meant to specify the behaviors of systems in a level of abstraction enough for reasoning and performing certain operations on them; namely, refinement and composition. Interfaces essentially specify how systems can be used while components specify what systems can do. Components hold a different set of properties from interfaces. They support the abstraction as well as composition operations. Interfaces and components can be related through implementation relations.

From a different perspective, components are meant to work in all environments, i.e. they are *input universal*, and do not constrain their environments. They *behave* or *mis-behave* in all environments. Interfaces, on the other hand, require a helpful environment to work in. In this way, environment plays an important role for interfaces. A helpful environment, which makes an interface work, would suffice for well-formedness of that in-
interface, i.e. they are input existential. In other words, they have some input assumptions about their environments and instead provide some output guarantees to the environment.

As an example, the “prototype” for a C function is an interface;\(^1\) any operation on a function prototype should observe the fact that only appropriate types for the signature of a function are meaningful. The body of a C function, in contrast, is a component. It can be called in different settings and even passing wrong parameters to a function body is valid, although a fault will happen. This is as opposed to function prototypes where any improper parameter passing is essentially meaningless. Assuming that we specify interfaces/components using logical predicates, following predicate is a component:\(^2\)

\[ \forall (x, y) \exists z (x \in \mathbb{R} \land y \in \mathbb{R}\{0\} \Rightarrow z = x/y) \quad (2.1) \]

On the other hand, this predicate is an interface:

\[ \forall (x, y) \exists z (x \in \mathbb{R} \land y \in \mathbb{R}\{0\} \land z = x/y) \quad (2.2) \]

These two predicates clearly demonstrate how components do not have any assumptions about their environments while interfaces do. In predicate 2.1, all values for \(x\) and \(y\) that are in \(\mathbb{R}\) are accepted. Furthermore, those values not belonging to \(\mathbb{R}\) are also accepted, they only cause a false implication which makes the whole predicate true. On the other hand, in predicate 2.2, there are certain restrictions on the value of \(x\) and \(y\) and ignoring those restrictions would make the predicate false.

Interface automata [dAH01a], is an automata-based interface for modelling component-based systems. We find this model appropriate for modelling Web services’ behavior; we will see more about this model in Section 3.1. Each state in an interface automaton can only accept certain inputs from the environment. Any inappropriate input can not be handled and is ignored. Interface automata’s syntax is quite similar and influenced by input-output automata [LT87], a component model, where each of its states is receptive to all possible inputs. Both interface automata and input-output automata belong to the stateful class of systems that we explore in Chapter 3.

---

\(^1\)Example extracted from [dAH01b].

\(^2\)The following two predicates are extracted from [dAH01b].
2.1. INTERFACE THEORIES

The rest of this chapter is organized as follows: In Section 2.1, we briefly discuss the basic concepts of interface theories, as described in [dAH01b], which are necessary to understand the rest of thesis, and then would focus on some interface models which we are more interested in, namely, family of port-dependency interfaces [dAH01b].

We introduce two new interface models of our own, Stateless Multiple Port Dependency Interface, in Section 2.2, and Enhanced Stateless Port Dependency Interface, in Section 2.3. We find these models useful for formalizing certain aspects of software systems.

2.1 Interface Theories

In the previous section we saw how interfaces are different from components:

- Interfaces specify how systems can be used while components specify what they do.
- Components work in all environments and thus are considered pessimistic, ready for all kinds of inputs at all states. On the other hand, interfaces only work in helpful environments and are considered optimistic, they assume that environment would provide their required inputs and accept their outputs.

In interface theories [dAH01b], the authors refine and formalize the above well-formedness criteria more precisely and introduce basic operations that can be performed on interfaces/components. Block diagrams are used to model interfaces/components. Blocks are defined by their ports, a set of typed variables, and the connections defined among those ports. It is often desirable to compose interfaces/components. Also, it is essential to realize whether two interfaces, or two components, behave similarly, i.e., one of them refines or abstracts the other. There could exist a sequence of refinement/abstraction relations among different blocks. In fact, it is possible to have a tree of refinement/abstraction relations among sets of blocks and compositions of blocks (which is a block itself). This creates a hierarchical block diagram with a hierarchy relation between different nodes.

Hierarchical block diagrams support refinement and abstraction relations. Abstraction lets a block diagram, probably consisting of multiple blocks, to be compressed into a single block while refinement expands a single block into a block diagram possibly consisting of
several blocks. Moving down the hierarchy provides us with more refined specification of a block while moving up provides us with more abstract specification of a block diagram. A more refined version of a block has weaker input assumptions and provides stronger output guarantees, see Figure 2.1.

“Weaker input assumptions” and “stronger output guarantees” are realized differently in different models. In general, “weaker input assumptions” refer to situations where the existing set of inputs are acceptable in a system with weaker input assumptions. “Stronger output guarantees”, on the other hand, promises that the system with stronger output guarantees at least satisfies the existing set of outputs and may even provide more.

As an example, consider an interface in form of a C function “prototype”, \( \text{int } \text{func}(\text{long}, \text{long}) \). A more refined version of this function should provide “stronger output guarantees” and “weaker input assumptions”. \( \text{short } \text{int } \text{func}(\text{double}, \text{double}) \) is a refinement of that prototype. In this example, we have manifested “stronger output guarantees” as more specific outputs and “weaker input assumptions” as being able to accept more inputs, i.e. the assumption on the type of inputs has become weaker.

The hierarchy relation affects components and interfaces differently. Recall that components essentially do not have any input assumptions; hence abstraction makes their outputs weaker and in effect the whole component weaker. In this way, components are meant to support bottom up verification [dAH01b]. In other words, in a tree of hierarchical relations among components, we should be able to move in a bottom-up fashion with higher levels being the abstraction of its corresponding lower level blocks. To make this issue more concrete in the context of compositional component systems, suppose \( f' \) and \( g' \) to be two compatible components and \( f \) and \( g \) be abstract versions of \( f' \) and \( g' \) respectively, then \( f \) and \( g \) will also be compatible. This is called compositional abstraction [dAH01b], and demonstrates a very interesting property of component-based systems, i.e. the capability of reasoning about compatibility and well-formedness of a component system by investigating properties of individual components independently. In the above example we only cared about the hierarchy relations among constituent components of a system, inherent properties of components enable us not to worry about the behavior of the whole system.

Conversely, interfaces support top-down design or compositional refinement [dAH01b]. Suppose \( F \) and \( G \) are two compatible interfaces then if \( F' \) and \( G' \) refine \( F \) and \( G \) respec-
Abstraction

Weaker Input assumptions
Stronger Output Guarantees

Stronger Input assumptions
Weaker Output Guarantees

Refinement

Figure 2.1: Hierarchy relation works contravariantly on input assumptions and output guarantees. A more refined version of a block has weaker input assumptions and stronger output guarantees. On the other hand, moving up the hierarchy provides more abstract version of a system which has stronger input assumptions and weaker output guarantee.
tively, then \( F' \) and \( G' \) are also guaranteed to be compatible. Again, notice how reasoning about constituent interfaces of a system can be carried out independently.

Note that interface theories only specify properties of interfaces and components but not the models themselves. In other words, interface theories only provide well-formedness criteria for certain category of models, i.e. interfaces and components. In the next subsection we formalize the attributes of such models in the context of block algebra formalism.

### 2.1.1 Interfaces/Components Formally

As mentioned, block algebra can be used to model interfaces and components. A block algebra consists of the following [dAH01b]:

- A set of blocks.
- For each block \( F \), a set of ports, \( P_F \). Each port, \( x \in P_F \), is a typed variable with type \( T_x \). Port names are assumed to be distinct. In other words, similar port names imply referring to the same port. In composability criteria for blocks, we show how similar port names are treated when two blocks are supposed to be composed.

- **Composition** is a commutative, associative partial binary function\(^3\) on a set of blocks. Given two composable\(^4\) blocks \( F \) and \( G \), the composition is defined as a new block, \( F \parallel G \).

- **Connection** is a partial binary function mapping a block \( F \) and an interconnect \( \theta \) to a block \( F\theta \). An interconnect is a set of channels where each channel is an ordered pair of ports, \((x, y)\), such that \( x \) is the source and \( y \) is the target of channel and \( T_x = T_y \). By using connections, we can redirect the value of a certain port to another port. In later sections, when we introduce different models, we show how connection relations can be defined for each model.

---

\(^3\)“A binary function, or function of two variables, is like a function, except that it has two inputs instead of one” (From Wikipedia, the free encyclopedia). A partial binary function is then a binary function which is not necessarily defined for all values of input.

\(^4\)Composability is a criteria that specifies which pairs of blocks can be composed.
2.1. INTERFACE THEORIES

$I_\theta$ and $O_\theta$ are respectively sets of source and target ports for an interconnect $\theta$:

$$I_\theta = \{x|\exists y)(x, y) \in \theta\}$$

$$O_\theta = \{y|\exists x)(x, y) \in \theta\}$$

$P_{F\theta}$ can then be defined as $(P_F \cup I_\theta \cup O_\theta)$.

- *Hierarchy* is a binary relation on a set of blocks, if $F' \preceq F$ then block $F'$ *refines* $F$ and block $F$ *abstracts* block $F'$.

Having defined the necessary notations, now we can formally define the properties of those block diagrams that represent interfaces and components.

**Interface**

A block algebra is an interface if:

- Given interfaces $F$, $F'$ and $G$ such that $F$ and $G$ are composable and $F' \preceq F$ then $(F'\parallel G) \preceq (F\parallel G)$, and

- Given interfaces $F$ and $F''$, such that $F' \preceq F$ and a connection is defined between $F$ and $\theta$, then $F'\theta \preceq F\theta$.

These properties express the top-down design nature of interfaces. By starting from an initial block diagram, it is possible to refine each of its constituent block diagrams into more refined versions without worrying about the outcome of the system as a whole. Interface theories guarantee that the outcome is compatible with the initial specification. In sections 2.1.2 and 2.1.3, we will study some examples of such systems.

**Component**

A block algebra is a component if:

- Given components $f'$, $g'$ and $f$ such that $f'$ and $g'$ are composable and $f' \preceq f$ then $(f'\parallel g') \preceq (f\parallel g')$, and
• Given components \( f' \) and \( f \), such that \( f' \preceq f \) and a connection is defined between \( f' \) and \( \theta \), then \( f'\theta \preceq f\theta \).

These properties suggest that verification of a set of components can happen independently and in a bottom-up fashion. In this thesis we are mainly interested in interfaces.

### 2.1.2 Stateless Input/Output Interface

In this section we present a simple interface model, *stateless input/output interface* \([\text{dAH01b}]\), which is capable of representing input/output behaviors of systems in a simple intuitive manner. Stateless interfaces, as opposed to stateful interfaces, do not represent the internal behaviors of systems. As an example, function prototypes can be considered stateless interfaces.

A stateless input/output interface \( F = \langle I_F, O_F, O_F^+ \rangle \), consists of a set of input ports, \( I_F \), a set of output ports, \( O_F \), and a set of available ports, \( O_F^+ \supseteq O_F \), which is disjoint from \( I_F \). Also, \( P_F = I_F \cup O_F \) and \( P_F^+ = I_F \cup O_F^+ \) are defined. Set of available ports, \( O_F^+ \), is the super set for output ports.

As you will see later in the refinement relation of stateless input/output interfaces, the refined version of such an interface may have more output ports than the the original interface. But, the refined version has a set of available ports that is not bigger than the original interface. This guarantees that the more refined interfaces can only use those output ports already defined in the set of available ports.

The key operations necessary to be defined for each interface model are composition and hierarchy. This model, similar to other models introduced in this chapter, satisfy the properties of interface models.

• **Composition:** composition is basically the union of involved interfaces’ ports. Two interfaces \( F \) and \( G \) are composable if port names do not overlap, i.e. \( P_F^+ \cap P_G^+ = \emptyset \). If \( F \parallel G \) is defined then, \( I_F \parallel_G = I_F \cup I_G \), \( O_F \parallel_G = O_F \cup O_G \) and \( O_F^+ \parallel_G = O_F^+ \cup O_G^+ \).

• **Connection:** Interconnect \( \theta \) and the interface \( F \) is defined if \( I_\theta \subseteq O_F \), \( O_\theta \cap O_F^+ = \emptyset \) and also the channels of such an interconnect can not have similar targets, meaning
that the value of each port can always be identified deterministically. $F\theta$ is then defined as following:

- $I_{F\theta} = I_F \setminus O_\theta$
- $O_{F\theta} = O_F \cup O_\theta$
- $O^{+}_{F\theta} = O^{+}_F \cup O_\theta$

In this way, it is only possible to create connections that originate from output ports, $O_F$. Furthermore, it is not possible to have interconnections where the source and destination ports both belong to output ports. Figure 2.2 represents a stateless input/output interface connected to an interconnect $\theta$ that consists of two channel.

- **Refinement**: $F'$ refines $F$, i.e. $F' \preceq F$, if:
  
  - $I_{F'} \subseteq I_F$
  - $O^{+}_{F'} \subseteq O^{+}_F$
  - $O_{F'} \supseteq O_F$

Note that the refined block can use less input ports but it has to provide at least all output ports of the original block. This contravariance relation, as mentioned earlier, is essential to interface models. *Interface automata*, described in Section 3.1, another interface model expressed through automata model, also suggests a contravariant refinement relation on its set of inputs and outputs but in a reverse order.

### 2.1.3 Stateless Port-Dependency Interface

*Stateless port dependency interfaces* [dAH01b], are *stateless input/output interfaces* with a set of dependencies defined between input and output ports. Formally, $F = \langle I_F, O_F, O^{+}_F, \kappa_F \rangle$, stateless port dependency interface (*PD Interface* henceforth), consists of a stateless input/output interface $\Pi_F = \langle I_F, O_F, O^{+}_F \rangle$ and a set of I/O dependencies $\kappa_F \subseteq I_F \times O_F$.

Any dependency pair belonging to $\kappa_F$ implies a dependency between an input and output port.
CHAPTER 2. INTERFACE MODELS

In this thesis, we represent block diagrams as a box with ports on the left representing input ports and ports on the right representing output ports. Interface $F = \langle \{a, b\}, \{c, d\}, \{c, d\} \rangle$ is connected to interconnect $\theta = \{(d, e), (c, a)\}$. $F\theta = \langle \{b\}, \{a, c, d, e\}, \{a, c, d, e\} \rangle$ is illustrated by dashed lines. Notice that since $a$ is the destination of a channel is removed from input ports and is added to the set of output ports.

The key operations necessary to be defined for each interface model are composition and refinement. This model is a well-formed interface model that satisfies the properties of interfaces.

- **Composition:** $F \parallel G$ is defined if $\Pi_F \parallel \Pi_G$ is defined and then, $\kappa_{F \parallel G} = \kappa_F \cup \kappa_G$.

- **Connection:** PD connections can be defined only if they do not result in dependency loops. Suppose there exists a dependency relation $(a, b)$ in the interface and there also exists a channel in the interconnect from output port $b$ to input port $a$. Then, $(a, b)$ is not a valid dependency since essentially $a$ and $b$ are connected already and thus dependent.

Dependencies for $F\theta$ are defined through dependencies defined in $F$. Consider $L = (\kappa_F \cup \theta)$, where $L \subseteq P_{F\theta} \times O_{F\theta}$, then $\kappa_{F\theta} = L^* \cap (I_{F\theta} \times O_{F\theta})$.\(^5\) In other words, if $L^*$ is transitive closure of $L$.\(^5\)

---

\(^5\) $L^*$ is transitive closure of $L$.\(^5\)
there exists any dependency relation implied through a connection relation, it should be present in $F\theta$. As an example, consider PD $F$ in Figure 2.3, $F$ has a dependency pair $(a, d) \in \kappa_F$ and, on the other hand, $d$ is connected to $e$. This implies that $(a, e) \in \kappa_{F\theta}$, as illustrated in Figure 2.3.

- **Refinement:** $F' \preceq F$ if $\Pi_{F'} \preceq \Pi_F$ and also $\kappa_{F'}$ does not suggest more dependencies on set of $\Pi_{F'}$ ports than $\Pi_F$ does. In other words, $\kappa_{F'} \cap (I_F \times O_F) \subseteq \kappa_F$.

In Figure 2.3, $F'$ refines $F$. $F'$ has less input ports and more output ports than $F$. Furthermore, $F'$ has a single dependency pair that is not on $F$ ports and thus does not violate refinement criteria.

### 2.1.4 Weak Interfaces

Interface definition, presented in section 2.1.1, suggests a compositional approach for software development. Notice that composition on any specific interface model should be both commutative and associative. While having these two properties for composition provides the chance for composing a set of interfaces in any arbitrary order, in some situations it could be restrictive. Sometimes sequential composition of interfaces suffices and there is no need for associativity to hold for such models. Those models are not well-formed interface models. However, they provide all well-formedness properties except associativity in composition.

We define a **weak interface** as a model which supports all criteria for being an interface (mentioned in 2.1.1) except the **associativity** in composition. Using weak interfaces, we can define more expressive composition operations ignoring the associativity. In this way, decomposition of a weak interface results in an ordered set of its constituent interfaces.

In the next section, we first introduce a new interface model, **Stateless Multiple Port Dependency Interface**, which is based on PD model and provides some new features for specifying port dependencies. Then in Section 2.3, we will introduce a weak interface, called **Enhanced Stateless Port Dependency Interface**, where port dependencies follow an order instead of being a set of unordered port dependencies. We formalize both models and prove they are an interface and a weak interface respectively.
Figure 2.3: $F$ has a connection over $\theta = \{(d, e)\}$. $F\theta$ is a PD and has an extra port $e$ and an extra dependency pair $(a, e)$. $(a, e)$ is defined since $d$ and $e$ are connected. $F'$ refines $F$; it has less input ports and more output ports than $F$ and its only dependency pair is defined over the port which does not belong to $F$ and thus is valid.
2.2 Stateless Multiple Port Dependency Interface

Stateless multiple port dependency interfaces, (MPDs), are an extension to PDs specifying multiple sets of dependencies on ports of an interface. Each of such dependency sets is a dependency scenario consisting of a set of dependency pairs. Multiple dependency scenarios enable to specify different independent behaviors of a certain system independently. In MPDs, each dependency pair can be either between an input and an output port or between an output and an input port. This is as opposed to PDs where only dependencies between input and output ports can exist. Such sets of dependencies are helpful. We show in Chapter 6 how this expressiveness is useful for specifying requests in a Web service discovery system. Web services can initiate an operation by sending an output and later receive an input message, related to that output message. Being able to specify output-input dependency pairs, we can model such scenarios. Below, we introduce this model formally.

Definition 2.1 An MPD interface $F = \langle I_F, O_F, O_F^+, H_F, U_F \rangle$, consists of a stateless input/output interface $\Pi_F = \langle I_F, O_F, O_F^+ \rangle$, and:

- $H_F$, the set of reserved ports, disjoint from $P_F^+$, which contains ports that cannot be used by $F$ or any MPD refining $F$. $P_F^\oplus$ is then equal to $(P_F^+ \cup H_F)$.
- $U_F = \{u_F^1, u_F^2, \ldots, u_F^p\}$, the set of dependency scenarios, where each $u_F^i \in U_F$ is a set of distinct dependency pairs and for each dependency pair $(a, b) \in u_F^i$, the following conditions are true:
  - $(a, b) \subseteq ((I_F \times O_F) \cup (O_F \times I_F))$
  - $\forall c \in P_F \Rightarrow \exists (b, c) \in u_F^i$
  - $\forall d \in P_F \Rightarrow \exists (d, b) \in u_F^i \cdot (d \neq a)$

Each dependency set represents one of the ways an interface could be used. Notice that this definition implies that if a dependency pair is in the transitive closure of a dependency set, $(a, b) \in (u_F^i)^*$ then its reverse pair can not be in the same set, $(b, a) \notin (u_F^i)^*$. In fact, the dependency pairs belonging to a dependency set do not form a loop. Hence it can be observed that in this model $(u_F^i)^* = u_F^i$. 
2.2.1 Connection

MPD connections can be defined only if channels originate from the output ports and the destination of all channels are distinct ports.

MPD connections, as opposed to PD connections, are defined over a set of interfaces. In fact, each MPD would only use its corresponding channels, i.e. channels initiating from its output ports or ending in its input ports. This is different from PDs where each PD is associated with its own interconnect. Our approach in MPD provides a broader view of a certain system and lets us design the interconnection between different interfaces in a more intuitive manner.

Another important issue is that interconnects can connect only output ports of an interface to input ports of some other interfaces. This is as opposed to PDs where we are able to connect output ports of an interface to its own input ports.

Formally, interconnect $\theta$ is defined over MPD $F = (I_F, O_F, O_F^+, H_F, U_F)$ if all of the following conditions hold:

\begin{itemize}
  \item $(I_\theta \cap P^\oplus_F) \subseteq O_F$, meaning channels, related to $F$, can initiate only from $F$’s output ports.
  \item $(O_\theta \cap P^\ominus_F) = \emptyset$, meaning channels can not end in $F$’s output ports.
  \item $\forall (x, y), (x', y') \in \theta \cdot (x \neq x') \Rightarrow y \neq y'$, meaning $\theta$ does not map two distinct ports to the same target port.
\end{itemize}

If interconnect $\theta$ is defined over $F$ then we can define $F\theta = (I_{F\theta}, O_{F\theta}, O_{F\theta}^+, H_{F\theta}, U_{F\theta})$ as following:

\begin{itemize}
  \item $I_{F\theta} = I_F$
  \item $O_{F\theta} = (O_F \cup \{ y | (x, y) \in \theta \wedge x \in O_F \}) \setminus (I_\theta \cap O_F)$
  \item $O_{F\theta}^+ = (O_F^+ \cup \{ y | (x, y) \in \theta \wedge x \in O_F \}) \setminus (I_\theta \cap O_F)$
  \item $H_{F\theta} = H_F \cup (I_\theta \cap O_F)$
  \item For each dependency rule $u_i^j \in U_F$, a dependency rule $u^j_{F\theta} \in U_{F\theta}$ is created as below:
\end{itemize}
2.2. STATELESS MULTIPLE PORT DEPENDENCY INTERFACE

1. \( L = u_F^i \cup (\theta \cap (O_F \times O_\theta)) \)
2. \( u_{F\theta} = L^* \cap (P_{F\theta} \times P_{F\theta}) \)

Before defining the concept of sharing an interconnect by different MPDS, we need to define a notation for the set of shared ports between two MPDs.

**Definition 2.2** Given two MPDs \( F \) and \( G \), \( \text{SharedPorts}(F,G) = P_F^+ \cap P_G^+ \) is the set of shared ports between \( F \) and \( G \).

\(\square\)

Now we can describe the well-formedness criteria for a mutual interconnect which is shared among multiple MPDs.

**Definition 2.3** A set of MPDs \( M = \{m_1, m_2, \ldots, m_n\} \) are connected to the same interconnect \( \theta \), or share a mutual interconnect \( \theta \), if the following conditions are true:

- For each \( a \in \text{SharedPorts}(m_i, m_j) \), such that \( m_i, m_j \in M \), either:
  - \( (a \in O_{m_i}) \wedge (a \in I_{m_j}) \wedge (\exists (b, a) \in \theta \cdot b \in H_{m_i}) \), or
  - \( (a \in O_{m_j}) \wedge (a \in I_{m_i}) \wedge (\exists (b, a) \in \theta \cdot b \in H_{m_j}) \)

- \( \forall m_k, m_l \in M \cdot a \in \text{SharedPorts}(m_k, m_l) \Rightarrow ((k = i) \wedge (l = j)) \vee ((k = j) \wedge (l = i)) \)

\(\square\)

2.2.2 Refinement

Given two MPD interfaces \( F = \langle I_F, O_F, O_F^+, H_F, U_F \rangle \) and \( F' = \langle I_{F'}, O_{F'}, O_{F'}^+, H_{F'}, U_{F'} \rangle \), \( F' \preceq F \), if all of the following conditions hold:

- \( \Pi_{F'} \preceq \Pi_F \) (\( \Pi_{F'} \) and \( \Pi_F \) are input/output components of \( F' \) and \( F \) respectively)

- \( H_{F'} \subseteq H_F \)
By replacing $F$ with $F'$, the interconnect that $F$ has been connected to, suppose it is $\theta$, remains a valid interconnect for $F'$ and all other MPDs that share that interconnect.

For each $u^i_{\theta_F} \in U_{F'}$, there exists $u^j_{\theta_F} \in U_F$ such that the following is true:

$$(u^i_{\theta_F} \setminus u^j_{\theta_F}) \cap ((I_F \times O_F) \cup (O_F \times I_F)) = \emptyset$$

The above restriction states that $F'$ can not have more dependency pairs on ports that it shares with $F$. However, extra dependency pairs on ports not belonging to $F$ are acceptable.

$F'$ may have less input ports and more output ports than $F$. Moreover, $F'$ can not introduce further dependencies on its set of common ports with $F$. However, it can specify dependencies on those ports of itself that are distinct from $F$.

We also require $F'$ to be connected and use the same connection that $F$ uses. This is since by refinement definition, we should be able to replace an existing interface with a more refined version of it. To replace $F$ with $F'$, it is crucial that $F'$ provides and rely on the same connection channels that $F$ provides; otherwise, inconsistencies may arise among other interfaces connected to $F$.

### 2.2.3 Composition

As an interface model, composition of MPDs is a commutative and associative binary function mapping two MPDs into a new MPD. Composition of two MPDs are defined only if they are composable.

Before formally defining composition in MPD, we define the following needed concepts and notations.

**Definition 2.4** Two MPDs $F = \langle I_F, O_F, O^+_F, H_F, U_F \rangle$ and $G = \langle I_G, O_G, O^+_G, H_G, U_G \rangle$ are composable if all of the following conditions hold:

- $F$ and $G$ use the same interconnect, suppose it is $\theta$.
- $H_F \cap H_G = \emptyset$
• For each \( u_i^F \in U_F, u_j^G \in U_G \) and \( L = u_i^F \cup u_j^G \), the transitive closure of \( L \), \( L^* \), the following is true:

\[
\forall (x, y) \in L^* \Rightarrow (y, x) \notin L^*
\]

The above restriction implies that by combining different dependency scenarios of two MPDs, they will not end up in a circular sequence of dependencies.

For two composable MPDs then, their composition can be defined as follows.

**Definition 2.5** Composition of two composable MPDs, \( F = \langle I_F, O_F, O^+_F, H_F, U_F \rangle \) and \( G = \langle I_G, O_G, O^+_G, H_G, U_G \rangle \), where \( U_F = \{u_1^F, u_2^F, \ldots, u_n^F\} \) and \( U_G = \{u_1^G, u_2^G, \ldots, u_m^G\} \), represented as \( F \parallel G = \langle I_{F\parallel G}, O_{F\parallel G}, O^+_{F\parallel G}, H_{F\parallel G}, U_{F\parallel G} \rangle \), can be computed as following:

• \( I_{F\parallel G} = (I_F \cup I_G) \setminus \text{SharedPorts}(F,G) \)

• \( O_{F\parallel G} = (O_F \cup O_G) \setminus \text{SharedPorts}(F,G) \)

• \( O^+_{F\parallel G} = (O^+_F \cup O^+_G) \setminus \text{SharedPorts}(F,G) \)

• \( H_{F\parallel G} = H_F \cup H_G \cup \text{SharedPorts}(F,G) \)

• \( U_{F\parallel G} = \bigcup u_{(i,j)}^{(F,G)} \) and \((1 \leq i \leq n), (1 \leq j \leq m)\), where each \( u_{(i,j)}^{(F,G)} \) is defined as following:

\[
u_{(i,j)}^{(F,G)} = (u_i^F \cup u_j^G)^* \setminus \text{((SharedPorts}(F,G) \times P^+_{F\parallel G}) \cup (P^+_{F\parallel G} \times \text{SharedPorts}(F,G)))
\]
A1 and A2 are two MPDs which are composable. Figure 2.5 shows how two MPDs are connected through an interconnect. We represent each dependency scenario for a MPD by using a different format of representation. For example, A2 has two dependency scenarios represented by dashed and dotted lines.

\[ \text{SharedPorts}(A1, A2) = \{e, b\} \]

there exists an appropriate channel in interconnect \( \theta \). Furthermore, \( H_{A1} \cap H_{A2} = \emptyset \) and different combinations of A1 and A2 dependency sets do not result cycles of dependencies.

The composition of A1 and A2 is shown in Figure 2.6. Notice how different dependency sets are combined and also how the resulting MPD does not contain any of the shared ports in its set of ports.

### 2.2.4 MPDs are interfaces

To show that MPDs are interfaces, we should prove that MPD complies with interface well-formedness criteria mentioned in Subsection 2.1.1. But before that, we first show that composition in MPD is both commutative and associative.
2.2. STATELESS MULTIPLE PORT DEPENDENCY INTERFACE

Figure 2.5: $A'1$ and $A'2$ are two MPDs that shared the same connection, $\theta = \{(x, e), (y, b)\}$. Figure 2.4 represents $A1$ and $A2$ which are equal to $A'1\theta$ and $A'2\theta$ respectively. For each member of the set of shared ports for $A1$ and $A2$, $\{e, b\}$, there exists an appropriate channel in $\theta$. Thus $A1$ and $A2$ are composable.

Figure 2.6: Composition of two MPDs in Figure 2.4. The set of dependencies for $A1 \parallel A2$ is defined as $\mathcal{U}_{A1\parallel A2} = \{(a, d), (a, g)\}, \{(a, d), (f, h)\}$. 
Theorem 2.1 For all MPDs $F$ and $G$, either they are not composable or $F \parallel G$ is defined and is equal to $G \parallel F$.

Proof. See the proof in Appendix A. □

Theorem 2.2 For all MPDs $F$, $G$ and $K$, either some of the MPDs are not composable or $(F \parallel G) \parallel K = F \parallel (G \parallel K)$.

Proof Sketch. See a proof sketch in Appendix A. □

Theorem 2.3 MPDs are interfaces.

Proof Sketch. See a proof sketch in Appendix A. □

2.3 Enhanced Stateless Port Dependency Interface

MPD interfaces are capable of expressing “dependency scenarios” where the order of dependency pairs are not important. Sometimes we are interested in a model capable of expressing the order between dependency pairs too. In this section, we introduce what we call enhanced stateless port dependency interface, EPD interface henceforth, which is a weak interface model based on MPD model. EPDs have a different set of properties and are capable of expressing order between dependency pairs.

Definition 2.6 An EPD weak interface $F = \langle I_F, O_F, O_F^+, H_F, \mathcal{V}_F \rangle$, consists of a stateless input/output interface $\Pi_F = \langle I_F, O_F, O_F^+ \rangle$, and:

- $H_F$, the set of reserved ports, disjoint from $O_F^+$, which contains ports that can not be used by $F$ or any MPD refining $F$. $P_F^\ominus$ is then equal to $(P_F^+ \cup H_F)$.

- $\mathcal{V}_F = \{v^1_F, v^2_F, \ldots, v^p_F\}$, the set of dependency rules, where each $v^i_F \in \mathcal{V}_F$ a sequence (ordered set) of distinct dependency pairs and for each dependency pair $(a, b) \in v^i_F$, similar to MPDs, either:
2.3. ENHANCED STATELESS PORT DEPENDENCY INTERFACE

\[- (a, b) \subseteq ((I_F \times O_F) \cup (O_F \times I_F)) \text{ and} \]
\[- \forall c \in P_F \Rightarrow \not \exists (b, c) \in v^i_F \text{ and} \]
\[- \forall d \in P_F \Rightarrow \not \exists (d, b) \in v^i_F \cdot (d \neq a) \]

or

\[- ((a = b) \land (a \in P_F)) \land (\not \exists (a, c), (c, a) \in v^i_F \cdot c \neq a) \]

Each dependency rule represents a sequence of related port dependencies. Also, each rule suggests a specific business rule or usage scenario. Multiple dependency rules provide an expressive formalism where we can specify the sets of related dependencies along with their temporal order. In this way, each EPD may provide more than one usage scenario.

The first type of dependency pairs, in Definition 2.6, suggests dependencies between an input and output port or conversely between an output and input port. We restrict the dependencies in such a way to avoid circular sequence of dependencies through transitivity. In fact, if a certain port is used in a dependency rule as the source of dependency it can not then appear as the destination of another port dependency, in the same dependency rule. Conversely, if a port appears as the destination of a dependency it can not act as a source in the same dependency rule. However, ports can appear in a different role in another dependency rule. Moreover, a certain port can not be influenced by more than one port.

In the context of component-based software design, the above restrictions make sense. In conventional interface-based software models, as opposed to hardware component systems, we are not interested in sending the outputs of components back to the same component. Moreover, we will later see, in Section 6.1.1, how these restrictions are appropriate for the purpose of modelling user requests in Web services matching.

The second type of dependencies introduce identity dependencies for a certain port, either input or output. This may not seem an intuitive dependency, however, it provides a level of expressiveness that we find useful in Subsection 6.1.1 where we are interested in using EPDs to formalize user requests in software matching. Notice that if a port appears
in an identity dependency it can not appear in any other dependency pairs in the particular dependency rule that it has appeared as identity dependency.

Obviously, transitive closure on pairs of dependencies, in any $v^i_F$, does not result in any loop. In other words, if $(x, y)$ is in transitive closure of $v^i_F$ then $(y, x)$ is not. This prevents having circular sets of dependencies in a dependency rule.

We also constrain the connection relation in such a way to prohibit circular dependencies caused by a sequence of dependency relations and channels belonging to an interconnect.

Before formally defining EPD’s operations, we need to introduce some necessary notations and operations.

### 2.3.1 Some Definitions/Notations

In this section we first provide some simple definitions and operations for sequences (ordered sets) and then introduce the respect relation between dependency rules belonging to two EPDs. Furthermore, we show how an EPD can be decomposed into an EPD and an interconnect.

**Set Operator**

Given a sequence, $\text{set}(s)$ is then a set containing only members of $s$. As an example, consider $s = \langle a, b, c \rangle$, then $\text{set}(s) = \{a, b, c\}$.

**Sequence Operator**

Assuming $r$ is a set, $\text{sequence}(r)$ is a sequence containing only members of $s$ in an arbitrary sequence. As an example, consider set $r = \{a, b, c\}$, a possible sequence for $\text{sequence}(r)$, is $\langle b, a, c \rangle$.

**Position Operator**

Given a sequence $s = \langle s_0, s_1, \ldots, s_m \rangle$, and a value $v$, the position of $v$ in sequence $s$, $\text{Pos}_s(v)$, is a set of integer numbers, $\{\text{pos}_1, \text{pos}_2, \ldots, \text{pos}_m\}$, such that for all $\text{pos}_i$, where $0 \leq i \leq m$: $s_{\text{pos}_i} = v$.

As an example, consider $s = \langle a, b, c, a \rangle$, $\text{Pos}_s(a) = \{1, 4\}$.
Subsequence Relation

Suppose $s$ is a sequence. $s'$, subsequence of $s$, $s' \in s$, is a sequence which omits some members of $s$. As an example, consider $s = \langle a, b, c, a \rangle$ and $s' = \langle a, c \rangle$, then $s' \in s$.

Sequence Concatenation

Given two sequences $r$ and $s$, the binary operation, sequence concatenation, maps $r$ and $s$, to $r \oplus s$, a new sequence containing all and only members of $r$ and $s$. Furthermore, the resulting sequence starts with $r$ members first and then is continued with $s$ members while maintaining the original sequence of $r$ and $s$.

As an example, consider $r = \langle a, b, c \rangle$ and $s = \langle d, e, a \rangle$, then $r \oplus s = \langle a, b, c, d, e, a \rangle$.

Sequence Subtract

Suppose $s$ is a sequence and $t$ is a set (or sequence), then sequence subtract, $s \ominus t$, is defined as a binary relation between a sequence and a sequence (or set); the result is a sequence $u$ which contains those elements of $s$ that are members of $s$ but are not members of $t$ and $u \in s$.

As an example, consider $r = \langle a, b, c \rangle$ and $s = \langle d, e, a \rangle$, then $r \ominus s = \langle b, c \rangle$.

Domain/Range of a Sequence of Pairs

Suppose $s$, a sequence of pairs, $s = \langle (a_0, b_0), (a_1, b_1), \ldots, (a_n, b_n) \rangle$, then domain and range of $s$, Domain($s$) and Range($s$) respectively, are defined as following:

$$\text{Domain}(s) = \langle a_0, a_1, \ldots, a_n \rangle$$
$$\text{Range}(s) = \langle b_0, b_1, \ldots, b_n \rangle$$

Sequence Transitive Closure of a Sequence of Pairs

Suppose $s$, a sequence of pairs, $s = \langle (a_0, b_0), (a_1, b_1), \ldots, (a_n, b_n) \rangle$, then sequence transitive closure of $s$, $s^\circ$, is a set of pairs containing all members of $s$ plus extra members where each extra member is implied through a transitive sequence $r = \langle (v_0, u_0), (v_1, u_1), \ldots, (v_m, u_m) \rangle$, as specified in Transitive Sequence Function, such that $r \in s$. Furthermore $s \in s^\circ$. 
As an example, consider \( s = \langle (a, b), (d, e), (b, c), (e, f), (c, g) \rangle \), then \( s^\circ = \langle (a, b), (d, e), (b, c), (e, f), (c, g), (d, f), (a, g) \rangle \). There are two extra members in \( s^\circ \) that are not in \( s \), namely, \( (d, f) \) and \( (a, g) \). \( (d, f) \) is implied through \( (d, e) \) and \( (e, f) \).

Transitive Sequence Function

Suppose \( s \), a sequence of pairs, \( s = \langle (a_0, b_0), (a_1, b_1), \ldots, (a_n, b_n) \rangle \). Also, consider a pair \( (x, y) \in s^\circ \) then \( TranSeq_s(x, y) \), transitive sequence of \( (x, y) \) on \( s \), is defined as a binary function which maps \( (x, y) \) and \( s \) to a unique sequence of pairs, \( \langle (u_0, v_0), (u_1, v_1), \ldots, (u_m, v_m) \rangle \) such that all of the following conditions are true:

- \( \forall i \ (0 \leq i \leq m), \ (u_i, v_i) \in s \)

- \( u_0 = x \)

- \( v_m = y \)

- \( \forall i \ (0 \leq i < m), \ v_i = u_{i+1} \)

- \( \# (u_j, v_j) \in s \cdot (u_j = v_m) \land (j > m) \) (meaning that the longest chain of transitive relation is considered.)

As an example, consider \( s = \langle (a, b), (d, e), (b, c), (e, f), (c, g) \rangle \), then \( s^\circ = \langle (a, b), (d, e), (b, c), (e, f), (c, g), (d, f), (a, g) \rangle \) and \( TranSeq_s(d, f) = \{(d, e), (e, f)\} \).

First Relation in Transitive Sequence

Suppose \( s \), a sequence of pairs representing a transitive relation and \( (x, y) \in s^\circ \) and \( TranSeq_s(x, y) \), transitive sequence of \( (x, y) \) on \( s \) being \( \langle (u_0, v_0), (u_1, v_1), \ldots, (u_m, v_m) \rangle \), then, we define \( FirstPair_s(x, y) \), first relation in transitive sequence, as a binary function which maps \( (x, y) \) and \( s \) to \( (u_0, v_0) \).

As an example, consider \( s = \langle (a, b), (d, e), (b, c), (e, f), (c, g) \rangle \), then \( s^\circ = \langle (a, b), (d, e), (b, c), (e, f), (c, g), (d, f), (a, g) \rangle \), \( TranSeq_s(d, f) = \{(d, e), (e, f)\} \) and \( FirstPair_s(d, f) = \{(d, e)\} \). As another example, for \( (a, b) \) that is not created through a chain of transitive relations, \( FirstPair_s(a, b) = \{(a, b)\} \).
2.3. ENHANCED STATELESS PORT DEPENDENCY INTERFACE

Inserting a value to a sequence

Given a sequence $s = \langle s_0, s_1, \ldots, s_n \rangle$, a value $v$ and position $i$, $0 \leq i \leq n$, we define add value operation, $s \triangleleft v$ as a binary operation on a sequence and a value, which adds the value $v$ to the sequence $s$ at position $i + 1$. The resulting sequence contains all the previous members plus $v$. Furthermore, all members with position greater than $i$ are shifted one position forward.

As an example, consider $s = \langle (a, b), (d, e), (b, c), (e, f), (c, g) \rangle$, then $s \triangleleft_2 (x, y) = \langle (a, b), (d, e), (x, y), (b, c), (e, f), (c, g) \rangle$.

Respect Relation in EPDs

Given two EPDs $F$ and $F'$ and their mutual interconnect $\theta$, the respect relation states that the dependency rules of $F'$ should either be independent of the input and output ports of $F$, or if they are using $F$’s ports then $F'$ dependency rules should not specify more dependencies on those ports than $F$ has specified. Furthermore, we require those dependencies in $F$ that are on ports involved in interconnect $\theta$, to be maintained in $F'$ dependencies too. This is required since $F'$ should be able to satisfy all the expectations that other interfaces have had from $F$, and as such should preserve the same dependencies on connections that $F$ has.

Definition 2.7 Given two EPDs $F = \langle I_F, O_F, O_F^+, H_F, \mathcal{V}_F \rangle$ and $F' = \langle I_{F'}, O_{F'}, O_{F'}^+, H_{F'}, \mathcal{V}_{F'} \rangle$ which are connected to the same interconnect $\theta$, the dependency set of $F'$ respects the dependency set of $F$, $\mathcal{V}_{F'} \subseteq \mathcal{V}_F$, if for each $v^i_{F'} \in \mathcal{V}_{F'}$ only one of the following is true:

- set$(v^i_{F'}) \cap (P_F \times P_F) = \emptyset$, which means $v^i_{F'}$ is not introducing any dependencies on $F$’s ports.
- There exists a $v^j_F \in \mathcal{V}_F$ such that the following conditions are true:
  - set$(v^j_{F'} \triangleleft v^i_{F'}) \cap (P_F \times P_F) = \emptyset$
  - $(v^j_{F'} \triangleleft (v^i_{F'} \triangleleft v^j_{F})) \subseteq v^j_{F}$

Note that an empty interconnect is defined for two EPDs, if they do not have a connection.
\[ \forall (a, b) \in v_j^i \cdot (a \in O_\theta) \lor (b \in O_\theta) \Rightarrow (a, b) \in v_i^j. \]

This means \( v_i^j \) is not introducing more dependencies on \( F \)'s ports than \( v_j^i \) does and, furthermore, it maintains all of the dependencies on connection ports that \( v_j^i \) specifies.

As an example consider \( r = \langle (a, b), (c, d), (f, g) \rangle \) as the only dependency rule of EPD \( F \) and \( s = \langle (a, b), (m, n), (f, g), (p, q) \rangle \) as the only dependency rule of EPD \( G \). Also, assume that \( F \) and \( G \) are connected to interconnect \( \theta = \{ (b, y) \} \), then \( \mathcal{V}_G \subseteq \mathcal{V}_F \).

### Some More Definitions

**Definition 2.8** Given an EPD \( F \), it can be decomposed to \( L \), an EPD, and the interconnect, \( \theta \) on \( L \), such that \( F = L \theta \). \( L \theta \) is then called decomposition of \( F \) in absence of connections, \( L \) and \( \theta \) are still defined and \( L = F \) and \( \theta = \emptyset \).

**Definition 2.9** Given two EPDs \( F \) and \( G \), the set of shared ports between the two EPDs, \( \text{SharedPorts}(F, G) \), is defined as \( P_F^+ \cap P_G^+ \).

Similar to MPD, three main operations on block algebras, connection, refinement and composition, can be defined for EPD. In the following sections, we first formalize these operations and then later, in Subsection 2.4, show that EPD is actually a weak interface and complies with the criteria we have specified for weak interface models.

### 2.3.2 Connection

EPD connections, similar to MPD connections, adhere to connection models in which an interconnect is defined over a set of EPDs. As discussed earlier, in Section 2.2.1,
such connections tend to be helpful in specifying a broad view of interconnections among different EPDs.

EPD connections can be defined only if channels originate from the output ports of interfaces and the destination of all channels are distinct ports.

Similar to MPDs, interconnect $\theta$ is defined over EPD $F = \langle I_F, O_F, O_F^+, H_F, V_F \rangle$ if:

- $(I_\theta \cap P^\ominus_F) \subseteq O_F$
- $(O_\theta \cap P^\ominus_F) = \emptyset$
- $\forall (x, y), (x', y') \in \theta \cdot (x \neq x') \Rightarrow y \neq y'$

If $\theta$ is defined over $F$, then EPD $F\theta = \langle I_{F\theta}, O_{F\theta}, O_{F\theta}^+, H_{F\theta}, V_{F\theta} \rangle$ is defined. $F\theta$ is computed similar to MPDs, except for the computation of dependency rules.

- $I_{F\theta} = I_F$
- $O_{F\theta} = O_F \cup \{y|(x, y) \in \theta \land x \in O_F\}$
- $O_{F\theta}^+ = O_F^+ \cup \{y|(x, y) \in \theta \land x \in O_F\}$
- $H_{F\theta} = H_F \cup (I_\theta \cap O_F)$
- For each dependency rule $v^i_F \in V_F$, a dependency rule $v^i_{F\theta} \in V_{F\theta}$ can be created by carrying out the following steps:
  1. $L = v^i_F \oplus \text{sequence}(\theta)$
  2. $X = v^i_F$
  3. For each $(a, b) \in (L^\oplus \ominus v^i_F)$
     
     \[
     p = \text{Pos}_X(\text{FirstPair}_L(a, b)) \]

Observation 2.4 For any such pairs $(a, b)$, there exists exactly one unique transitive sequence. This is since channels belonging to interconnect $\theta$ can only initiate from $O_F$ and their destinations cannot be in $I_F$. Intuitively, the only path for creating such $(a, b)$ is by starting from a dependency pair $(a, y)$, where $y \in O_F$, and $y \in I_\theta$, and moving to a channel $(y, b) \in \theta$ and finally creating $(a, b) \in L^\oplus$. Observe that there is no other ways to create such $(a, b)$. 
(b) $X = X \boxplus_\rho (a, b)$

4. $v_{F_\theta}^i = X \ominus \{((a, b) \in X | (a \in I_\theta) \lor (b \in I_\theta))\}$

2.3.3 Refinement

Given two EPDs $F = \langle I_F, O_F, O^+_F, H_F, V_F \rangle$ and $F' = \langle I_{F'}, O_{F'}, O^+_{F'}, H_{F'}, V_{F'} \rangle$, $F'$ refines $F$, $F' \preceq F$, if the following conditions are true:

- $\Pi_{F'} \preceq \Pi_F$
- $H_{F'} \subseteq H_F$
- $V_{F'} \subseteq V_F$.

Intuitively, $F'$ may have less input ports and more output ports than $F$. Furthermore, $H_{F'}$ should be subset of $H_F$; this is to make composability of $F'$ with other EPDs more likely.

Moreover, $F'$ can not introduce further dependencies on its set of common ports with $F$, i.e. any dependency rule in $F'$ defined over $F$ ports should respect at least one of $F$’s dependency rules. However, it can specify dependencies on its ports that are distinct from $F$.

We also require $F'$ to be connected and use the same interconnect that $F$ uses. This is since, by the refinement definition, we should be able to replace an existing interface with a more refined version of it. To replace $F$ with $F'$, it is crucial that $F'$ provides and relies on the same connection channels that $F$ provides, otherwise inconsistencies may arise among other weak interfaces that are connected to $F$.

2.3.4 Composition

We define composition between two EPDs as a function which maps two composable EPDs to a new EPD. The resulting EPD combines the involved EPDs’ ports and dependency rules. EPD dependency rules are created by considering different combinations of two EPDs’ dependency rules. According to weak interface definition, EPD composition suffices to be commutative only.
Shared ports between the two involved EPDs are removed from the resulting composition. This is since the shared ports of involved EPDs are results of connections and once we compose the two EPDs the channels between the two EPDs become internal part of the resulting composition EPD and can be removed from the set of input and output ports.

**Definition 2.10** Two EPDs $F = \langle I_F, O_F, O_F^+, H_F, V_F \rangle$ and $G = \langle I_G, O_G, O_G^+, H_G, V_G \rangle$ are composable if all of the following conditions hold:

- $F$ and $G$ use the same interconnect, suppose it is $\theta$.
- $H_F \cap H_G = \emptyset$
- For each $v_i^F \in V_F$, $v_j^G \in V_G$ and $L = \text{set}(v_i^F) \cup \text{set}(v_j^G)$, $L^*$,\(^8\) the transitive closure of $L$:
  \[
  \forall (x, y) \in L^* \Rightarrow (y, x) \notin L^*
  \]

Composability between two EPDs suggests that they are connected via the same connection and are properly connected through their ports. Furthermore, different combinations of two EPDs’ dependency rules do not imply any circular dependencies. Moreover, two composable EPDs should not have any reserved ports in common. This intuitively makes sense because reserved ports represent the set of ports which have been involved in a connection or previous compositions. Thus, two EPDs which have some common reserved ports, suggest that they have some elements embedded inside them which makes their composition meaningless.

As an example, consider two EPDs in Figure 2.7, $B1 = \langle I_{B1}, O_{B1}, O_{B1}^+, H_{B1}, V_{B1} \rangle$, where $I_{B1} = \{a, b\}$, $O_{B1} = \{c, d, e\}$, $O_{B1} = O_{B1}^+$, $H_{B1} = \{x\}$ and $V_{B1} = \{(a, d), (a, e)\}$ and also, $B2 = \langle I_{B2}, O_{B2}, O_{B2}^+, H_{B2}, V_{B2} \rangle$ where $I_{B2} = \{e, f\}$, $O_{B2} = \{g, h, b\}$, $O_{B2} = O_{B2}^+$, $H_{B2} = \{y\}$ and $V_{B2} = \{(e, g), (f, h), (f, b)\}$. To check whether these two EPDs are composable, they should satisfy the composability conditions mentioned in Definition 2.10.

\(^8\)Notice that this is simple transitive closure operation rather than sequence transitive closure,\(^5\), used so far.
Figure 2.7: Two Composable EPDs. The numbers over dependencies represent the order of dependency pairs for a certain dependency rule.

In Figure 2.8, we provide the decomposition of two EPDs shown in Figure 2.7. As you can see the two EPDs are connected through the same interconnect $\theta$ and this satisfies the first condition of composability. Also, the set of shared ports, $\{e, b\}$, participate in interconnect $\theta = \{(d, e), (h, b)\}$, as required by composability definition. Secondly, $H_{B1}$ and $H_{B2}$ are disjoint sets which satisfies the second condition of composability. And finally, it can be observed that different combinations of dependency rules in $B1$ and $B2$ do not result circular dependencies. As an example, we show how $v^1_{B1}$ and $v^1_{B2}$ observe that criteria. Following, we compute $L^*$ as required in Definition 2.10.

$$L = set(\{(a, d), (a, e)\}) \cup set(\{(e, g)\})$$

$$L^* = \{(a, d), (a, e), (a, g)\}$$

Notice that $L^*$ does not result any circular dependencies. Similarly, other combinations of dependency rules can be checked to be valid. Thus $B1$ and $B2$ are composable.

Before defining composition between two EPDs, we define a relation necessary for
2.3. ENHANCED STATELESS PORT DEPENDENCY INTERFACE

Figure 2.8: Two EPDs $B'1$ and $B'2$ and their mutual interconnect $\theta = \{(d, e), (h, b)\}$. $B'1\theta$ and $B'2\theta$ are respectively decompositions of $B1$ and $B2$ shown in Figure 2.7.

defining the composition. *Alternating transitive closure* is a binary operation mapping two sequences of dependency pairs to a single dependency pair. The resulting pair is created through transitivity of pairs belonging to the two sequences. Following definitions makes this concept more precise.

**Definition 2.11** Given two nonempty sequences $r = \langle (a_0, b_0), (a_1, b_1), \ldots, (a_m, b_m) \rangle$ and $s = \langle (u_0, v_0), (u_1, v_1), \ldots, (u_n, v_n) \rangle$, two sequences of dependency pairs, alternating transitive closure of $r$ and $s$, $AltTran(r, s)$, is a binary operation mapping $r$ and $s$ to a dependency pair $(x, y)$. Furthermore, there exists an alternating sequence, transitive chain of relations, $AltTranSeq(r, s) = \langle (x_0, y_0), (x_1, y_1), \ldots, (x_p, y_p) \rangle$, disjoint from $AltTran(r, s)$, such that:

$$AltTran(r, s) = \begin{cases} 
(x_0, x_0) & \text{if } x_p = y_p \\
(x_0, y_p) & \text{if } x_p \neq y_p
\end{cases}$$
\textit{AltTransSeq}(r, s) = ⟨(x_{0}, y_{0}), (x_{1}, y_{1}), \ldots, (x_{p}, y_{p})⟩ \text{ starts from } (a_{0}, b_{0}), \text{ first element of } r, \text{ and the following are true about its elements:}

\begin{itemize}
  \item ∀ i \ (0 \leq i < p) \cdot (y_{i} = x_{i+1})
  \item ∀ i \ (0 \leq i \leq p) \cdot (i = 2k) \Rightarrow \exists (a_{j}, b_{j}) \in r \cdot (x_{i}, y_{i}) = (a_{j}, b_{j})
  \item ∀ i, j \ (0 \leq i, j \leq p) \cdot (i = 2k) \land (j = 2k') \land (i < j) \Rightarrow (\text{Pos}_{r}(x_{i}, y_{i}) < \text{Pos}_{r}(x_{j}, y_{j}))
  \item ∀ i \ (0 \leq i \leq p) \cdot (i = 2k + 1) \Rightarrow \exists (u_{j}, v_{j}) \in s \cdot (x_{i}, y_{i}) = (u_{j}, v_{j})
  \item ∀ i, j \ (0 \leq i, j \leq p) \cdot (i = 2k + 1) \land (j = 2k' + 1) \land (i < j) \Rightarrow (\text{Pos}_{s}(x_{i}, y_{i}) < \text{Pos}_{s}(x_{j}, y_{j}))
  \item p = 2k \Rightarrow \#(a_{i}, b_{i}) \in s \cdot (y_{p} = a_{i}) \land (i > \text{Pos}_{s}(x_{p-1}, y_{p-1}))
  \item p = 2k + 1 \Rightarrow \#(u_{i}, v_{i}) \in r \cdot (y_{p} = u_{i}) \land (i > \text{Pos}_{r}(x_{p-1}, y_{p-1}))
\end{itemize}

As an example consider two sequences of dependency pairs, \(r = ⟨(a, b), (c, d), (e, f), (g, h)⟩\) and \(s = ⟨(l, m), (b, e), (n, o), (f, p)⟩\), then \textit{AltTrans}(r, s) = (a, p) \text{ and } \textit{AltTransSeq}(r, s) = ⟨(a, b), (b, e), (e, f), (f, p)⟩.

Intuitively, alternating transitive closure of two sequences \(r\) and \(s\) is created by an alternating sequence of pairs starting from the first element of \(r\) and alternatively moving forward between \(s\) and \(r\), until either there is no more relation in \(r\) or transition can not be continued in one of \(r\) or \(s\). Notice that if there does not exist any sequence of transitive pairs, the first element of \(r\) will be the alternating transitive closure of \(r\) and \(s\).

\textbf{Definition 2.12} \textit{Given two nonempty sequences } \(r = ⟨(a_{0}, b_{0}), (a_{1}, b_{1}), \ldots, (a_{m}, b_{m})⟩\) \text{ and } \(s = ⟨(u_{0}, v_{0}), (u_{1}, v_{1}), \ldots, (u_{n}, v_{n})⟩\), \textit{their alternating transitive closure } \textit{AltTrans}(r, s) = ⟨x, y⟩ \text{ and their transitive chain of relations, } \textit{AltTransSeq}(r, s) = ⟨(x_{0}, y_{0}), (x_{1}, y_{1}), \ldots, (x_{p}, y_{p})⟩, \textit{we define non transitive sequence of relations on } r \text{ and } s, \textit{AltNonTransSeq}(r, s) = ⟨(q_{0}, t_{0}), (q_{1}, t_{1}), \ldots, (q_{i}, t_{i})⟩, \text{ such that:}
2.3. ENHANCED STATELESS PORT DEPENDENCY INTERFACE

- AltNonTranSeq(r, s) = (\bigoplus_{i=0}^{p-1} (\rho_i)) \setminus \text{AltTranSeq}(r, s)
- \rho_0 = \langle (u_0, v_0), (u_1, v_1), \ldots, (u_{\text{Pos}(x_1,y_1)-1}, v_{\text{Pos}(x_1,y_1)-1}) \rangle
- \forall i \ (1 \leq i \leq p) \cdot (i = 2k) \Rightarrow \rho_i = \langle (u_{\text{Pos}(x_{i-1},y_{i-1})+1}, v_{\text{Pos}(x_{i-1},y_{i-1})+1}), \ldots, \\
  (u_{\text{Pos}(x_{i+1},y_{i+1})-1}, v_{\text{Pos}(x_{i+1},y_{i+1})-1}) \rangle
- \forall i \ (1 \leq i \leq p) \cdot (i = 2k + 1) \Rightarrow \rho_i = \langle (a_{\text{Pos}(x_{i-1},y_{i-1})+1}, b_{\text{Pos}(x_{i-1},y_{i-1})+1}), \ldots, \\
  (a_{\text{Pos}(x_{i+1},y_{i+1})-1}, b_{\text{Pos}(x_{i+1},y_{i+1})-1}) \rangle

As an example consider two sequences of dependency pairs, \( r = \langle (a, b), (c, d), (e, f), (g, h) \rangle \) and \( s = \langle (l, m), (b, e), (n, o), (f, p) \rangle \), then AltTran(r, s) = (a, p), AltTranSeq(r, s) = \langle (a, b), (b, e), (e, f), (f, p) \rangle and AltNonTranSeq(r, s) = \langle (l, m), (c, d), (n, o) \rangle.

Before, presenting composition definition in EPDs, we define an attribute of dependency rules that helps us to define the composition.

**Definition 2.13** Given two composable EPDs \( F \) and \( G \) and their mutual interconnect \( \theta \), a dependency rule \( v_F^i \in v_F \), is closed with respect to \( G \) if for all \( (a, b) \in v_F^i \), \( a, b \notin O_\theta \).

**Definition 2.14** Composition of two composable EPDs, \( F = \langle I_F, O_F, O_F^+, H_F, V_F \rangle \) and \( G = \langle I_G, O_G, O_G^+, H_G, V_G \rangle \), where \( V_F = \{v_F^1, v_F^2, \ldots, v_F^n\} \) and \( V_G = \{v_G^1, v_G^2, \ldots, v_G^n\} \), and \( F \) and \( G \) are connected through interconnect \( \theta \), is represented as \( F \parallel G = \langle I_F||G, O_F||G, O_F^+||G, H_F||G, V_F||G \rangle \) and can be computed similar to MPD composition (except for the dependency rules) and is defined as following:

- \( I_F||G = (I_F \cup I_G) \setminus \text{SharedPorts}(F,G) \)
- \( O_F||G = (O_F \cup O_G) \setminus \text{SharedPorts}(F,G) \)
- \( O_F^+||G = (O_F^+ \cup O_G^+) \setminus \text{SharedPorts}(F,G) \)
• $H_{F\parallel G} = H_F \cup H_G \cup SharedPorts(F, G)$

• $\mathcal{V}_{F\parallel G} = \mathcal{V}_{(F,G)} \cup \mathcal{V}_{(G,F)} \cup \mathcal{V}_{(F,F)} \cup \mathcal{V}_{(G,G)}$

where $\mathcal{V}_{(F,G)} = \bigcup v^{(i,j)}_{(F,G)}$, $\mathcal{V}_{(G,F)} = \bigcup v^{(j,i)}_{(G,F)}$, $\mathcal{V}_{(F,F)} = \bigcup v^{(i,i)}_{(F,F)}$, $\mathcal{V}_{(G,G)} = \bigcup v^{(j,j)}_{(G,G)}$ and $(1 \leq i \leq n), (1 \leq j \leq m)$.

Each $v^{(i,j)}_{(F,F)} \in \mathcal{V}_{(F,F)}$, has one of the following values:

- $v^i_F$: If $v^i_F$ is closed with respect to $G$ and all $v^j_G \in \mathcal{V}_G$ are not closed with respect to $F$.
- $\emptyset$: Otherwise.

Similarly, each $v^{(j,j)}_{(G,G)} \in \mathcal{V}_{(G,G)}$, has one of the following values:

- $v^j_G$: If $v^j_G$ is closed with respect to $F$ and all $v^i_F \in \mathcal{V}_F$ are not closed with respect to $G$.
- $\emptyset$: Otherwise.

Each $v^{(i,j)}_{(F,G)} \in \mathcal{V}_{(F,G)}$, can be computed by first computing the dependencies using the algorithm shown in Figure 2.9.

$$v^{(i,j)}_{(F,G)} = \text{ComputeDependency}(v^i_F, v^j_G, SharedPorts(F, G))$$

Similarly, each $v^{(j,i)}_{(G,F)} \in \mathcal{V}_{(G,F)}$, can be computed as following:

$$v^{(j,i)}_{(G,F)} = \text{ComputeDependency}(v^j_G, v^i_F, SharedPorts(F, G))$$

Algorithm in Figure 2.9 receives two dependency rules belonging to two EPDs and computes their alternating transitive closure. Once the computation is carried out, the alternating transitive and non-transitive sequences that are used to generate such an alternating transitive closure pair are removed from the set of dependencies belonging to the two dependency rules. The algorithm continues computing alternating transitive closure of the new sequences until one of the sequences becomes empty.
Procedure ComputeDependency(sequence $v^i_F$, sequence $v^i_G$;
set SharedPorts($F, G$));

Variables: sequence

$$Temp_F = v^i_F,$$
$$Temp_G = v^i_G,$$
$$Result, \nu, alt\_tran, alt\_non\_tran;$$

begin
while ($Temp_F \neq ()$) and ($Temp_G \neq ()$) do
$$\nu = sequence(Alt\_Tran(r, s)) ;$$
$$alt\_tran = Alt\_TranSeq(r, s) ;$$
$$alt\_non\_tran = Alt\_Non\_TranSeq(r, s) ;$$
$$Result = Result \oplus alt\_non\_tran ;$$
$$Result = Result \oplus \nu ;$$
$$Temp_F = ((Temp_F \oplus alt\_tran) \oplus alt\_non\_tran) \oplus \nu ;$$
$$Temp_G = ((Temp_G \oplus alt\_tran) \oplus alt\_non\_tran) \oplus \nu ;$$
endw

$$Result = Result \oplus Temp_F ;$$
$$Result = Result \oplus Temp_G ;$$
if ($set(Domain(Result)) \cap SharedPorts(F, G) \neq ()$) \lor
($set(Range(Result)) \cap SharedPorts(F, G) \neq ()$) then
$$Result = sequence(\emptyset)$$
endif
return Result
end

Figure 2.9: The outline of the algorithm for computing a dependency rule from two dependency rules belonging to two composable EPDs.
Intuitively, alternating transitive sequences are generated through interconnect $\theta$ which connects the two EPDs. In this way, once we recognize a chain of connections, those ports being involved in those connections become useless in the composition of the two EPDs. In other words, we no longer consider them input and output ports because they are consumed internally by the two EPDs. Observe that the shared ports are not useful for any other EPDs other than the two EPDs involved in the composition. This is since, by connection definition, there could exist only one channel ending at a certain destination port. Once either of sequences becomes empty, we quit the loop and append the rest of the sequences remained in the two dependency rules.

Also, notice that the algorithm considers those resulting dependency rules that have a shared port in their domain or range, as invalid dependency rules. Such dependency rules suggest that there does not exist any synchronization plans that would provide the connections between the two EPDs to work in synchrony. Thus, those dependency rules are considered invalid and are removed.

The composition of two composable EPDs in Figure 2.7, can be computed as mentioned above. Figure 2.10 shows the resulting EPD which has only one dependency rule. Notice that the algorithm would compute some dependency rules for $B_1 \parallel B_2$ that have dependencies on shared ports, which are invalid. The only valid dependency rule is generated by composing dependency rule $\langle (a, d), (a, e) \rangle$ belonging to $F$ with $\langle (e, g) \rangle$ belonging to $G$.

In the next section we will show that EPDs are in fact weak interface models and follow well-formedness criteria for weak interfaces. But before showing that, we define some useful operations on EPDs that can be used to express interactions between EPDs, which can
2.3. ENHANCED STATELESS PORT DEPENDENCY INTERFACE

not be specified through composition.

2.3.5 Some Useful Operations

In this subsection we introduce two binary functions on EPDs that express execution scenarios between two EPDs that can not be expressed by composition. Concatenation and choice represent the sequential and choice patterns of execution respectively. These two patterns are very useful and common patterns of execution; they both represent independent execution of two interfaces and inherently require that the two involved EPDs not to communicate through their ports; we make these conditions concrete in below definitions. Furthermore, the required condition guarantees that these operations result in well-formed EPDs.

Definition 2.15 Concatenation of two EPDs, \( F = \langle I_F, O_F, O_F^+, H_F, V_F \rangle \) and \( G = \langle I_G, O_G, O_G^+, H_G, V_G \rangle \), where \( V_F = \{ v_{F}^{1}, v_{F}^{2}, \ldots, v_{F}^{n} \} \) and \( V_G = \{ v_{G}^{1}, v_{G}^{2}, \ldots, v_{G}^{m} \} \), is an EPD, represented as \( FG = \langle I_{FG}, O_{FG}, O_{FG}^+, V_{FG} \rangle \), and is defined if for each \( x \in \text{SharedPorts}(F,G) \), either \( (x \in I_F) \land (x \in I_G) \) or \( (x \in O_F^+) \land (x \in O_G^+) \). Then \( FG \) can be computed as following:

- \( I_{FG} = I_F \cup I_G \)
- \( O_{FG} = O_F \cup O_G \)
- \( O_{FG}^+ = O_F^+ \cup O_G^+ \)
- \( H_{FG} = H_F \cup H_G \)
- \( V_{FG} = \bigcup v_{(F,G)}^{(i,j)} \) and \((1 \leq i \leq n), (1 \leq j \leq m)\).

Each \( v_{(F,G)}^{(i,j)} \in V_{(F,G)} \) is a well-formed dependency rule, as required in Section 2.3, and is computed as following:

\[
v_{(F,G)}^{(i,j)} = v_{F}^{i} \oplus v_{G}^{j}
\]

Concatenation operation is different from composition in the sense that it avoids concurrency and only provides sequential execution of two EPDs. In Section 7.2.1 we will
see how this operation can be used to express interactions between EPDs that can not be expressed by composition.

Definition 2.16 Choice of two EPDs, \( F = \langle I_F, O_F, O^+_F, H_F, \mathcal{V}_F \rangle \) and \( G = \langle I_G, O_G, O^+_G, H_G, \mathcal{V}_G \rangle \), where \( \mathcal{V}_F = \{v^1_F, v^2_F, \ldots, v^n_F\} \) and \( \mathcal{V}_G = \{v^1_G, v^2_G, \ldots, v^m_G\} \), is an EPD represented as \( F \parallel G = \langle I_{F \parallel G}, O_{F \parallel G}, O^+_{F \parallel G}, H_{F \parallel G}, \mathcal{V}_{F \parallel G} \rangle \), and is defined as following if for each \( x \in \text{SharedPorts}(F, G) \), either \( (x \in I_F) \wedge (x \in I_G) \) or \( (x \in O^+_F) \wedge (x \in O^+_G) \).

- \( I_{(F \parallel G)} = I_F \cup I_G \)
- \( O_{(F \parallel G)} = O_F \cup O_G \)
- \( O^+_{(F \parallel G)} = O^+_F \cup O^+_G \)
- \( H_{(F \parallel G)} = H_F \cup H_G \)
- \( \mathcal{V}_{(F \parallel G)} = \mathcal{V}_F \cup \mathcal{V}_G \).

Intuitively, \( F \parallel G \) at any time would execute either \( F \) or \( G \). Similar to concatenation, this operation provides a certain level of expressivity unavailable in composition.

### 2.4 EPDs Are Weak Interfaces

To show that EPDs are weak interfaces, we first show that composition in EPD is a commutative operation and then would show how EPDs comply with other well-formedness criteria for interfaces.

**Theorem 2.5** For all EPDs \( F \) and \( G \), either they are not composable or \( F \parallel G \) is defined and is equal to \( G \parallel F \).

**Proof Sketch.** See a proof sketch in Appendix A. \( \square \)

**Theorem 2.6** Enhanced Stateless Port Dependency Interface is a weak interface model.
Proof Sketch. See a proof sketch in Appendix A.

In Chapter 6, we show how we can use EPD as a model for capturing the requests of users in a Web service discovery system.

2.5 Discussion

In this chapter, we introduced two stateless interface models that are capable of expressing dependencies between inputs and outputs of systems. EPD, a weak interface model, tends to specify the temporal order of such dependencies. On the other hand, MPD is a less expressive model but more flexible in carrying out composition operations. Associativity along with commutativity, as exists in MPD composition, is an elegant property enabling us to carry out composition on a set of MPDs in any arbitrary order. In other words, true parallelism for an operation can be achieved in presence of commutativity and associativity. EPD, on the other hand, only supports commutativity on composition operation and thus requires that the composition on a number of interfaces to be carried out in a sequential order.

While we prefer to have a model as expressive as EPD and as flexible as MPD, it is not possible to have both properties together. The reason being that, in EPD, we are using the concept of sequences of dependencies and composition between two EPDs properly interleaves different dependency scenarios. Such interleaving, however, does not consider all possible alternating sequences. Instead, it only generates the natural interleaving of such dependencies. By natural interleaving, we mean interpreting each dependency pair as a synchronous relation and interpreting a dependency rule as a single thread of execution. Such an interleaving turns out to be non-associative; to have associativity in EPD composition, we should in fact keep necessary information about all possible interleavings between two EPDs dependency rules. In the context that we are going to use EPDs, this is rather unnecessary. In fact, interface automata, which are stateful interfaces introduced in next chapter, are capable of handling such comprehensive sets of interleavings.

Nevertheless, it becomes obvious how MPD and EPD complement each other’s capabilities in different respects and how they can be useful in different contexts. As an example, consider the composition of two MPDs in Figure 2.6, on page 27, and composition of two
EPDs in Figure 2.10, page 44. Observe how MPDs generate extra unnecessary dependency sets, where as, if each involved MPDs’ dependency sets were specified precisely, as specified as EPDs in Figure 2.7, we would have a smaller and more accurate result, i.e. EPD in Figure 2.10. This implies that if we have more specific information about port dependencies, we would better use EPDs rather than MPDs. Later in Chapter 6, we will see how differently these two models can be useful in discovery of Web services.
Chapter 3

Stateful Interfaces

Stateful interfaces are meant to model more dynamic aspects of systems and characterize behaviors of systems at run time. Although interfaces are essentially meant to provide information about how we can use a certain system, stateful interfaces provide even some information about what systems do and what they can provide as outputs. However, stateful interfaces remain interface models and comply with well-formedness criteria defined for interface models.

In this chapter, we first explain interface automata [dAH01a] which are automata-based interface models and then introduce our own stateful model, Parameterized Interface Automata, on top of it.

The remainder of this chapter is organized as follows: in Section 3.1, interface automata is introduced with some extensions to its formalism that we require in this thesis. In Section 3.2, we introduce our interface automata-based formalism, Parameterized Interface Automata, which is an enhanced model meant to model complex actions\(^1\) in interface automata. And finally, sections 3.3, 3.4 and 3.5 explore the composition, refinement and inherent properties of parameterized interface automata in details.

\(^1\)Complex actions are actions comprised of some simple actions.
3.1 Interface Automata

In the previous chapter, we discussed stateless interfaces. Stateful interfaces are different from stateless interfaces in the sense that they provide information about the behavior of an interface in run time. Interface Automata is a stateful automata-based interface inspired by input-output automata. Interface automata tends to be a useful model for modelling the characteristics of component-based systems. Similar to all interface models, it supports top-down refinement; also, it introduces an intuitive composition method. Usually, operations such as composition, on state-based formalisms, lead to what is called “state explosion”. The composition in interface automata, on the other hand, is very efficient in this respect. This is partly because the size of interface automata tend to be small since they are not meant to be responsive to all actions in all states and partly because concepts such as shared actions and illegal states tend to minimize the size of resulting compositions.

An interface automaton, IA henceforth, is an automaton machine consisting of a set of states (with one initial state), disjoint sets of input, output and internal actions and a set of transitions. Each transition happens between two distinct states through one action. IAs are nondeterministic machines which means from one state, by applying the same actions, we may move to different states. Non-determinism is a valuable property of IAs. We found it an essential property for modelling workflow patterns (see Section 4.7).

IAs are meant to model different aspects of software systems. Input and output actions can be interpreted differently; in [dAH01a], the authors suggest that input actions can be used to model: procedures and methods that can be called and the return point of such calls. Also, input actions can be assumed as the receiving end of communication channels. Output actions, on the other hand, can model procedure or method calls and the act of termination or returning from such calls. Furthermore, they can represent the act of sending a message.

In Section 3.2, we introduce parameterized interface automata which is capable of modelling complex actions, consisting of multiple simple parameters. As such, parameterized interface automata can be used to model procedure and method calls more precisely and

2Interface automata and I/O automata aim at modelling two different concepts though. I/O automata specifies components (it is input universal) while interface automata models interfaces (it is input existential). However, they syntactically resemble each other.
efficiently than interface automata.

### 3.1.1 Interface Automata Formally

$P = (V_P, V_P^{\text{init}}, A_I^P, A_O^P, A_H^P, \tau_P)$ represents an IA where $V_P$ is the finite set of states and $V_P^{\text{init}} \subseteq V_P$ is the set of initial states with at most one state. $A_I^P, A_O^P$ and $A_H^P$ are disjoint sets of input, output and internal\(^3\) actions respectively. $\tau_P$ is a set of steps between states such that $\tau_P \subseteq V_P \times A_P \times V_P$, where $A_P = A_I^P \cup A_O^P \cup A_H^P$.

Each step $(v, a, v')$, is an input step if $a \in A_I^P$, and respectively output or internal step if $a \in A_O^P$ or $a \in A_H^P$. For each state $u$, $A_I^P(u), A_O^P(u)$ and $A_H^P(u)$ are respectively the sets of input, output and internal actions enabled in that state; i.e. the set of actions with which there is a step initiating from $u$.

An execution fragment, $\langle v_0, a_0, v_1, a_1, \ldots, v_n \rangle$, of $P$, is a finite sequence of states and actions such that for all $0 \leq i < n$, $(v_i, a_i, v_{i+1}) \in \tau_P$. For two states $v, u \in V_P$, $v$ is reachable from $u$, if there exists an execution fragment with its first state being $u$ and its last state being $v$. In general, state $v$ is said to be reachable if there exists a state $u \in V_P^{\text{init}}$ and $v$ is reachable from $u$. Operations such as composition may create some unreachable states as the result. We are not interested in unreachable states and remove them.

The set of shared actions between two IAs $P$ and $Q$ can be defined as:

$$\text{shared}(P, Q) = A_P \cap A_Q$$

Two IAs $P$ and $Q$ are composable if they do not take same inputs and do not produce same outputs and, furthermore, their internal actions do not overlap with other actions:

$$\begin{align*}
(A_I^P \cap A_I^Q) &= \emptyset \\
(A_O^P \cap A_O^Q) &= \emptyset \\
(A_H^P \cap A_Q) &= \emptyset \\
(A_H^Q \cap A_P) &= \emptyset
\end{align*}$$

\(^3\)When an input action is internally consumed by an output action or vice versa the two actions are reduced to an internal action.
3.1.2 Some Extensions to IA Formalism

**Definition 3.1** Schedule of an execution fragment $s = \langle v_0, a_0, v_1, a_1, \ldots, v_n \rangle$, defined as $\text{sched}(s)$, is the sequence of actions, $\langle a_0, a_1, \ldots, a_{n-1} \rangle$.

**Definition 3.2** Given an execution fragment of $P$, $s = \langle v_0, a_0, v_1, a_1, \ldots, v_n \rangle$, and $\text{sched}(s) = \langle a_0, a_1, \ldots, a_{n-1} \rangle$, $\text{InputSchedule}(s)$, $\text{OutputSchedule}(s)$ and $\text{InternalSchedule}(s)$, namely, input, output and internal subsequences of $s$ respectively, are functions mapping $s$ to a subsequence of $\text{sched}(s)$, $r = \langle b_0, b_1, \ldots, b_l \rangle$, such that $r \subseteq \text{sched}(s)$ and the following are true:

- $r = \text{InputSchedule}(s)$:
  
  \[(r \subseteq \text{sched}(s)) \land (\text{set}(r) = \text{set}(\text{sched}(s)) \cap A_I^P)\]

- $r = \text{OutputSchedule}(s)$:
  
  \[(r \subseteq \text{sched}(s)) \land (\text{set}(r) = \text{set}(\text{sched}(s)) \cap A_O^P)\]

- $r = \text{InternalSchedule}(s)$:
  
  \[(r \subseteq \text{sched}(s)) \land (\text{set}(r) = \text{set}(\text{sched}(s)) \cap A_H^P)\]

**Definition 3.3** A state $u \in V_P$ is a terminating state of IA, $P$, if there does not exist any steps initiating from that state. $V_P^{\text{term}}$ the set of terminating states is defined as follows:

\[V_P^{\text{term}} = \{v \mid (v \in V_P) \land (\not\exists (v, a, u) \in \tau_P)\}\]
3.1. INTERFACE AUTOMATA

Definition 3.4 A state \( u \in V_P \) is an iterating state of IA, \( P \), if it is not a terminating state and the destination of all transitions initiating from it are in the initial set of states. \( V_P^{\text{loop}} \) the set of iterating states can be defined as following:
\[
V_P^{\text{loop}} = \{ v \mid (v \in V_P) \land (v \notin V_P^{\text{term}}) \land (\forall (v, a, u) \in \tau_P \Rightarrow u \in V_P^{\text{init}}) \}.
\]

Definition 3.5 An execution fragment, \( \langle v_0, a_0, v_1, a_1, \ldots, v_n \rangle \), of an IA, \( P \), is an execution if:
\[
(v_0 \in V_P^{\text{init}} \land v_n \in (V_P^{\text{init}} \cup V_P^{\text{term}}))
\]
And the set \( \text{Exec}(P) \) is the finite set of all possible executions in \( P \). Furthermore \( \text{Scheds}(P) \) is the set of schedules for such executions.

Definition 3.6 Given two interface automata \( P \) and \( Q \), if:
\[
\text{shared}(P, Q) = (A_P^I \cap A_Q^I) \cup (A_P^O \cap A_Q^O) \cup (A_P^H \cap A_Q^H)
\]
(3.1)

Concatenation on two IAs \( P \) and \( Q \), represented as \( PQ \), is a binary function resulting a new IA. It models the sequential execution of two IAs; after \( P \) has finished its execution and reached one of its terminating states, \( Q \) starts executing from its initial state. Formally:
\[
\begin{align*}
V_{PQ} &= (V_P \cup V_Q) \setminus V_P^{\text{term}} \\
V_{PQ}^{\text{init}} &= V_P^{\text{init}} \\
A_{PQ} &= A_P \cup A_Q \\
\tau_{PQ} &= \{(v, a, u) \mid (v, a, u) \in \tau_P \land u \notin V_P^{\text{term}}\} \cup \\
&\quad \{(v, a, u) \mid (\exists (v, a, u') \in \tau_P \cdot u' \in V_P^{\text{term}}) \land u \in V_Q^{\text{init}}\} \cup \\
&\quad \{(v, a, u) \mid (\exists (v, a, u') \in \tau_P \cdot v \in V_P^{\text{loop}} \land u' \in V_Q^{\text{init}}) \land u \in V_Q^{\text{init}}\} \cup \\
&\quad \{(v, a, u) \mid (v, a, u) \in \tau_Q\}
\end{align*}
\]
The restriction imposed, in equation 3.1, is to guarantee the well-formedness of resulting concatenated IAs.

Definition 3.7 Given two interface automata $P$ and $Q$, if:

$$\text{shared}(P, Q) = (A^I_P \cap A^I_Q) \cup (A^O_P \cap A^O_Q) \cup (A^H_P \cap A^H_Q)$$

Choice on two IAs $P$ and $Q$, represented as $P \times Q$, is a binary function resulting a new IA with an extra state $\iota$ which serves as mutual initial state of $P$ and $Q$. $P \times Q$ models the choice between execution of two IAs; either $P$ or $Q$ can be executed at one time. Formally:

$$V_{P \times Q} = (V_P \cup V_Q \cup \{\iota\}) \setminus (V_{\text{init}}^P \cup V_{\text{init}}^Q)$$
$$V_{P \times Q}^{\text{init}} = \{\iota\}$$
$$A_{P \times Q} = A_P \cup A_Q$$
$$\tau_{P \times Q} = \{(v, a, u) | (v, a, u) \in \tau_P \land v \notin V_{\text{init}}^Q \} \cup \{(v, a, u) | (v, a, u) \in \tau_Q \land v \notin V_{\text{init}}^P \} \cup \{(\iota, a, u) | \exists (v, a, u) \in \tau_P \cdot v \in V_{\text{init}}^P \} \cup \{(\iota, a, u) | \exists (v, a, u) \in \tau_Q \cdot v \in V_{\text{init}}^Q \}$$

Intuitively, the choice operation creates a new IA that is capable of providing the services that $P$ and $Q$ provide. However, at one time only $P$ or $Q$ can be used but not both simultaneously.

IA provides an intuitive graphic representation of software systems. In Figure 3.1 we present an IA which is in fact an IA representation of a complex Web service specified in [ea03]. This example mainly complies with message transmission interpretation of IA, however, as mentioned earlier IA can be used to model procedure/method calls too. In Section 4.7, we show how Web services’ descriptions can be mapped into their equivalent interface automata.

3.1.3 Composition

Composition of two IAs $P$ and $Q$, $P \parallel Q$ is defined based on the product of two IAs. The product of two composable IAs $P$ and $Q$, $P \otimes Q$, is an IA consisting of all possible interleaved
Figure 3.1: An IA representing a complex Web service. State “1” is the initial state and state “11” is a terminating state. Input, output and internal steps are represented by their action names followed by “?”, “!” and “;” respectively. The modelled Web service is meant to provide functionality of an online bookshop. By providing the “bookName” as input to the service, its ISBN is retrieved and automatically put in the shopping cart. “putInCartISBN” is an internal action, once the book name is searched and its ISBN is found, it is put in the shopping cart. If user of the system already has a user id, “signInInfo” is received and the service proceeds with receiving necessary information for payment through credit card. If user does not have a user id then a new account can be created. At the end, necessary information for arranging the delivery of the book is received. Note that this complex IA is comprised of some other IAs that are mainly executed in a sequential order, except for the sign in section where there is a choice to either sign in or create a new account.
execution scenarios of the two IAs, except for those actions belonging to $\text{shared}(P,Q)$. For shared actions, the product would synchronize the two IAs in appropriate states where the shared actions can possibly happen. At those states, the two actions, an input and an output, are reduced to a new internal action. Following is a more formal definition of the product for two composable IAs, $P$ and $Q$:

$$
V_{P \otimes Q} = V_P \times V_Q
$$

$$
V_{P \otimes Q}^{\text{init}} = V_P^{\text{init}} \times V_Q^{\text{init}}
$$

$$
A_{P \otimes Q}^I = (A_P^I \cup A_Q^I) \setminus \text{shared}(P,Q)
$$

$$
A_{P \otimes Q}^O = (A_P^O \cup A_Q^O) \setminus \text{shared}(P,Q)
$$

$$
A_{P \otimes Q}^H = A_P^H \cup A_Q^H \cup \text{shared}(P,Q)
$$

$$
\tau_{P \otimes Q} = \{(v,u), a, (v',u') \mid (v, a, v') \in \tau_P \land a \notin \text{shared}(P,Q) \land u \in V_Q\} \cup
\{(v,u), a, (v',u') \mid (u, a, u') \in \tau_Q \land a \notin \text{shared}(P,Q) \land v \in V_P\} \cup
\{(v,u), a, (v',u') \mid (v, a, v') \in \tau_P \land (u, a, u') \in \tau_Q \land a \in \text{shared}(P,Q)\}
$$

Figures 3.2 and 3.3 are two IAs representing two Web services. $Prod$ represents a Web service that searches for a product either by its name or its id; $Pay$ is an IA presenting a Web service that performs payments for credit cards. It receives a credit card number and an amount in Canadian dollar and processes the requested payment. In case the payment is carried out successfully, a reference number and otherwise an error code is returned back. Note that although $Pay$ has “us\_price” as its input action it does not have any step using that action. These types of actions could appear when two IAs are composed. We will later see how such actions can be generated through composition. Actions, such as “us\_price”, can potentially create illegal states in compositions. Figure 3.4 is the product of the two IAs. Many states in $Pay \otimes Prod$ become unreachable and are removed along with the steps associated with those actions. The black-filled states in Figure 3.4 are illegal states. Composition of two IAs is different from their product; it does not contain any illegal states.

Formally, illegal states of two IAs, $P$ and $Q$, are those states in which one of the IAs has a shared action enabled in that state and the other IA does not have any steps using
the corresponding action. Formally:

\[
\text{Illegal}(P, Q) = \left\{ (v, u) \in V_P \times V_Q \mid \exists a \in \text{shared}(P, Q) \cdot \left( a \in A^O_P(v) \land a \notin A^I_Q(v) \lor a \in A^O_Q(u) \land a \notin A^I_P(v) \right) \right\}
\]

Existence of illegal states can create problems. In the presence of a helpful environment \(^4\), it is possible to avoid reaching these states. An environment for an IA, \( P \), is itself an IA. \( E \) is an environment for \( P \) if all of the following conditions are true:

- \( E \) and \( P \) are composable.
- \( E \) is nonempty.
- \( E \) can receive all of \( P \)'s outputs, \( A^I_E = A^O_P \).
- \( \text{Illegal}(P, E) = \emptyset \).

\(^4\)Recall that interfaces rely on helpful environments
To avoid illegal states of two IAs $P$ and $Q$, an environment is required that never reaches those states. In other words, an environment $E$ is a legal environment for $P \otimes Q$ if no state in illegal($P, Q$) is reachable in $(P \otimes Q) \otimes E$.

Figure 3.5 represents an environment for $Prod \otimes Pay$. This environment avoids reaching illegal states of $Buyer \otimes Prod$ (see Figure 3.4). Those environments that never reach the illegal states of a pair of IAs are legal environments for those two IAs. Thus $Buyer$ is a legal environment for $Prod$ and $Pay$. In Figure 3.6, we have shown the product of $(Buyer \otimes Prod) \otimes Pay$, note that the black-filled states in Figure 3.4 are never reached by $Buyer$.

Composition of two IAs $P$ and $Q$, $P \parallel Q$, can be produced from their product. In fact, the composition of two IAs is their product limited only to their compatible states. Compatible states of two IAs, $P$ and $Q$, $Comp(P, Q)$, are defined if there exists an environment $E$ such that no illegal states can be reached from any of states in $Comp(P, Q)$. In other
Figure 3.4: This IA represents the result of $Prod \otimes Pay$. The states with the filled circle are illegal states.
Figure 3.5: Buyer is a legal environment for Prod ⊗ Pay (Figure 3.4).

Figure 3.6: Product of Buyer ⊗ Prod ⊗ Pay.
words,
\[ A_{P\parallel Q} = A_{P\times Q} \]
\[ V_{P\parallel Q} = \text{Comp} (P, Q) \]
\[ V_{P\parallel Q}^{\text{init}} = V_{P\times Q}^{\text{init}} \cap \text{Comp} (P, Q) \]
\[ \tau_{P\parallel Q} = \tau_{P\otimes Q} \cap (\text{Comp}(P, Q) \times A_{P\parallel Q} \times \text{Comp} (P, Q)) \]

In [dAH01a], authors show how the composition of two IAs can be constructed in linear time in \(|P| \times |Q|\).\(^5\) The idea of the composition algorithm introduced in [dAH01a] is to start from each illegal state \( u \) and move backward to those states that reach \( u \) by either an internal or output action and then for all such states, they are in turn checked for the same states that reach them in the same manner. By removing all illegal states and the states reaching them as mentioned above the composition of two IAs can be generated. See [dAH01a] for more details.

Figure 3.7 shows \( \text{Prod} \parallel \text{Pay} \). Notice that the illegal states in Figure 3.4 are removed. Since the only steps leading to the illegal states are input steps, "in_us?", we only have to remove the illegal state themselves, illustrated as a black-filled states in Figure 3.4.

### 3.1.4 Refinement

A more refined version of an IA at least provides the services that the IA provides. In IA, the refinement relation again acts contravariantly on input and output actions. A more refined IA can have more input actions but no more output actions than the original one.\(^6\) The IA in Figure 3.8, \( \text{GenPay} \), refines the IA in Figure 3.3; it not only provides all the functionalities provided by \( \text{Pay} \) but also lets the payments to happen through bank accounts, it has more input actions. On the other hand, it has less output actions and does not provide "err_no" as its output.

Internal actions are also involved in refinement definition. The formal definition is based on the refinement relation between two states of two IAs. By starting from the initial states

---

\(^5\)The size of an IA, \( P \), is \(|P| = |V_P| + |	au_P|\).

\(^6\)Note that the relation is as opposed to what we described in stateless interfaces where it was possible to have more outputs and less inputs. This difference is justified by different interpretation of input and output elements in stateless interfaces and IAs. In this thesis, however, we are not interested in applying stateless interfaces approach to refinement.
Figure 3.7: Prod $\parallel$ Pay.
of two IAs and propagating the refinement checking to other states, it is possible to check whether an IA, \( Q \), refines IA \( P \), \( P \succeq Q \).\(^7\) To accurately express this, we should first define some necessary notations.

For a state \( v \) in an IA \( P \), \( \varepsilon\text{-closure}_P(v) \), is a set containing \( v \) and all states that can be reached from \( v \) through internal steps. Then, all input and output actions initiating from such states (from \( \varepsilon\text{-closure}_P(v) \)) are respectively specified by their externally enabled input and output actions: \( \text{ExtEn}^I_P(v) \) and \( \text{ExtEn}^O_P(v) \). Formally,

\[
\text{ExtEn}^I_P(v) = \{ a \mid \forall u \in \varepsilon\text{-closure}_P(v) \cdot a \in \mathcal{A}^I_P(u) \}
\]

\[
\text{ExtEn}^O_P(v) = \{ a \mid \exists u \in \varepsilon\text{-closure}_P(v) \cdot a \in \mathcal{A}^O_P(u) \}
\]

Note that, input actions are meant to be enabled in all states in \( \varepsilon\text{-closure}_P(v) \); this is because an environment for an IA can not distinguish in which state of \( \varepsilon\text{-closure}_P(v) \) the IA currently is. Thus it is desirable that \( P \) accepts a specific input action in all of its states.

For a state \( v \) and an externally enabled \( a \in \text{ExtEn}^I_P(v) \cup \text{ExtEn}^O_P(v) \), externally

\(^7\) Notice that the direction of refinement symbol in IA is in the opposite direction of what we had in interface theories.
reachable states are defined as following:

\[
\text{ExtDest}_P(v, a) = \{ u' | \exists(u, a, u') \in \tau_P \cdot u \in \varepsilon\text{-}closure_P(v) \}
\]

As mentioned earlier, the refinement relation between two IAs is defined by checking states of two IAs. The binary relation *alternating simulation* \(\succeq \subseteq V_P \times V_Q\) between two states \(v \in V_P\) and \(u \in V_Q\) (represented as \(v \succeq u\)) holds if the following conditions are true:

- \(\text{ExtEn}_P^I(v) \subseteq \text{ExtEn}_Q^I(u)\)
- \(\text{ExtEn}_Q^O(u) \subseteq \text{ExtEn}_P^O(v)\)
- \(\forall a \in (\text{ExtEn}_P^I(v) \cup \text{ExtEn}_Q^O(u)), \forall u' \in \text{ExtDest}_Q(u, a) \Rightarrow \exists v' \in \text{ExtDest}_P(v, a) \cdot v' \succeq u'\)

This relation essentially defines the basics of a recursive refinement relation where a state in the refined IA can have more input actions and less output actions enabled than the original interface. Furthermore, all neighboring states in the refined IA should have a counterpart state in the original IA (the third condition of the relation.)

Based on the above relation, IA, \(Q\), refines IA, \(P\), \(P \succeq Q\), if:

- \(A_I^P \subseteq A_I^Q\)
- \(A_O^Q \subseteq A_O^P\)
- \(\exists v \in V_P^{\text{init}}, \exists u \in V_Q^{\text{init}} \cdot v \succeq u\)

Refinement has interesting properties; it is a reflexive and transitive relation and furthermore, it is compositional. As an example, IA in Figure 3.9, \((\text{Prod} \parallel \text{GenPay})\), refines the IA in Figure 3.7, \((\text{Prod} \parallel \text{Pay})\). This is true since \(\text{Pay} \succeq \text{GenPay}\). The refinement relation, \((\text{Prod} \parallel \text{Pay}) \succeq (\text{Prod} \parallel \text{GenPay})\), need not to be checked again since compositionality of refinement relation guarantees that. In other words, since \(\text{Pay} \succeq \text{GenPay}\), it is guaranteed that \((\text{Prod} \parallel \text{Pay}) \succeq (\text{Prod} \parallel \text{GenPay})\). In [dAH01a], the authors describe such properties of IAs in details.
3.1. INTERFACE AUTOMATA

Figure 3.9: IA ($Prod \parallel GenPay$). Note that this IA refines the composed interface automata in Figure 3.7.
3.2 Parameterized Interface Automata

In this section we introduce parameterized interface automaton (PIA), an interface model based on IA, which is capable of modelling software systems in a finer level of abstraction. This model mainly remains compatible with IA and thus is able to use useful properties of IA. The essential difference between the two models is the existence of complex actions in PIA. A complex action consists of more than one simple action. We believe that IA’s level of abstraction for modelling procedures and procedure calls is rather high. Actions in IA are meant to model software artifacts such as procedures, methods and messages. Such artifacts can have simple elements. Procedures and methods can have multiple parameters and messages can consist of different elements. We present PIA which helps us to model software systems in a more intuitive manner.

Complex actions in PIA represent functions, methods, procedures or complex messages. Each complex action, used in a step, can be decomposed into an execution fragment with a schedule only consisting of simple actions. As an example, Figure 3.10 shows a PIA. “pay_in_cnd” is a complex action of that automaton. This automaton is meant to provide similar functionality as IA in Figure 3.3 provides.

Complex actions either represent input or output behaviors and thus should either consist of input or output actions. Also, it is possible for complex actions to have internal actions as their constituent elements. In this way, it is possible that constituent simple actions of a certain complex action are consumed during different steps of compositions. In other words, values for different parameters of a function or different elements of a complex message can be provided by different PIAs. This is an intuitive model for component based and collaborative systems that require complicated communications.

Complex actions consisting of simple input actions, such as “pay_in_cnd” in Figure 3.10, can receive their required elements from different components. It can be assumed that a function call only happens once all the constituent simple actions are consumed by appropriate output actions.

As an example, “credit_no” and “cnd_price” belonging to “pay_in_cnd”, can individually be consumed by actions belonging to two different PIAs. E.g. the value for “credit_no” can

---

8Consumption, as described in IA, happens when an input and output action belonging to the set of shared actions are synchronized and reduced to an internal action.
Figure 3.10: PIA CompPay representing a similar service shown in Figure 3.3. Note that “pay_in_cnd” is a complex action and represents an execution fragment with a schedule consisting of simple actions “credit_no” and “cnd_price”.
be received from a searchProfile component while the value for “cnd_price” can be received from a searchProduct component. Supposing that “pay_in_cnd” simulates a function call, receiving values for its two parameters through two PIAs can be interpreted as buffering the first parameter as it is received and performing the function call once the second value is also received.

For complex actions consisting of simple output actions, it can be assumed that their simple output elements can be either consumed by different simple input actions or they could fit one or more input complex actions. The important issue is that, regardless of the type of complex actions, their constituent simple actions should not be interleaved by any other actions. We will make these notions more precise in the next section.

3.2.1 Parameterized Interface Automata Formally

A PIA, \( P = \langle V^{\text{abst}}_P, V^{\text{init}}_P, V^{\text{comp}}_P, A^{\text{I simp}}_P, A^{\text{O simp}}_P, A^{\text{H simp}}_P, A^{\text{comp}}_P, \tau^{\text{abst}}_P, \phi_P \rangle \), has the following elements:

- \( V^{\text{abst}}_P \): is a set of states called abstract states. In Figure 3.10, states 1, 3 and 4 are abstract states of CompPay. A step \((v, a, u)\) of \( P \) is an abstract step if both \( u \) and \( v \) belong to \( V^{\text{abst}}_P \). \((1, \text{pay in cnd}, 3)\) is an abstract step of \( P \).

- \( V^{\text{init}}_P \): is the set of states from which a PIA starts its execution and \( V^{\text{init}}_P \subseteq V^{\text{abst}}_P \). Similar to IA, we have maximum number of one initializing state. In Figure 3.10, state 1 is the initial state of CompPay.

- \( V^{\text{comp}}_P \): is a set of states called complex states which is disjoint from abstract states. In Figure 3.10 state 2 is a complex state of CompPay. A step \((v, a, u)\) of \( P \) is a complex step if either \( u \) or \( v \) or both belong to \( V^{\text{comp}}_P \). \((1, \text{credit no}, 2)\) is a complex step of \( P \).

Also, we let \( V_P = V^{\text{comp}}_P \cup V^{\text{abst}}_P \) and require each \( v \in V^{\text{comp}}_P \) not to have more than one step initiating from it.

- \( A^{\text{I simp}}_P, A^{\text{O simp}}_P \) and \( A^{\text{H simp}}_P \) are the sets of simple input, simple output and simple internal actions. The set of simple actions for \( P \), \( A^{\text{simp}}_P \) is \( A^{\text{I simp}}_P \cup A^{\text{O simp}}_P \cup A^{\text{H simp}}_P \). Each action belonging to \( A^{\text{simp}}_P \) can be used to define both abstract and complex steps.
3.2. PARAMETERIZED INTERFACE AUTOMATA

- $\mathcal{A}_P^{\text{comp}}$: is the set of complex actions where $\mathcal{A}_P^{\text{comp}} \cap \mathcal{A}_P^{\text{simp}} = \emptyset$. These actions can only be used to define abstract steps. We let $\mathcal{A}_P = \mathcal{A}_P^{\text{comp}} \cup \mathcal{A}_P^{\text{simp}}$.

- $\tau_P^{\text{abst}} \subseteq (V_P^{\text{abst}} \times \mathcal{A}_P \times V_P^{\text{abst}})$ is the set of abstract steps.

- $\phi_P$: Stepping function is an injective function on the set of abstract steps. It maps each step to an alternating sequence of states and simple actions. Before formally specifying the characteristics of stepping function, we need to define complex fragments.

Definition 3.8 An alternating sequence of states and actions, $s = \langle v_0, a_0, v_1, a_1, \ldots, v_n \rangle$, is a complex fragment for an abstract step $(v, a, u)$ if all of the following conditions are true:

- $(n > 1) \land (v_0 = v) \land (v_n = u)$
- $\forall i, j \cdot (i \neq j) \land (0 \leq i, j \leq n) \Rightarrow v_i \neq v_j$
- $\forall i \cdot (0 < i < n), \ v_i \in V_P^{\text{comp}}$
- $(\forall i \cdot (0 \leq i \leq n), \ a_i \in (\mathcal{A}_P^{I,\text{simp}} \cup \mathcal{A}_P^{H,\text{simp}})) \lor$
- $(\forall i \cdot (0 \leq i \leq n), \ a_i \in (\mathcal{A}_P^{O,\text{simp}} \cup \mathcal{A}_P^{H,\text{simp}}))$

Then execution schedule of such a complex fragment, $\text{sched}(s)$, will be $\langle a_0, a_1, \ldots, a_{n-1} \rangle$ and furthermore, the following is true about it:

$$\forall \ (x = (v, a, u), y = (v', b, u')) \in \tau_P^{\text{abst}} \cdot \text{sched}(\phi_P(x)) = \text{sched}(\phi_P(y)) \iff a = b$$

Having defined complex fragments, we can now define stepping function. For each abstract step $x = (v, a, u) \in \tau_P^{\text{abst}}$, $\phi_P(x)$ is defined as following:

$$\phi_P(x) = \begin{cases} 
  s = (v, a, u) & \text{if } a \in \mathcal{A}_P^{\text{simp}} \\
  s = \langle v_0, a_0, v_1, a_1, \ldots, v_n \rangle & \text{if } a \in \mathcal{A}_P^{\text{comp}} \text{ then } s \text{ is a complex fragment for } x 
\end{cases}$$

More than one step can not be mapped to the same sequence.
Furthermore, given two distinct abstract steps, \( x = (v, a, u) \) and \( y = (v', b, u') \) such that \( a, b \in A_{\text{comp}}^P \) and \( \phi(x) = (v_0, a_0, v_1, a_1, \ldots, v_n) \) and \( \phi(y) = (v'_0, b_0, v'_1, b_1, \ldots, v'_m) \), if \( A = \{v_1, v_2, \ldots, v_{n-1}\} \) and \( B = \{v'_1, v'_2, \ldots, v'_{m-1}\} \) then the following is true:

\[
A \cap B = \emptyset
\]

In fact, an abstract step can be viewed in two levels of abstraction, a simple abstract step or as an execution fragment (complex fragment) only consisting of simple actions. It is important to note that complex fragments can not be interleaved by some other actions and fragments and thus composition for PIAs would be different from IAs. Also, each abstract step is either comprised of entirely internal actions, combination of input and internal actions or combination of output and internal actions. This intuitively makes sense; if we consider an abstract step as a method or function, then it is evident that its constituent elements represent that method’s parameters and thus all of them should be input actions. Conversely, a function call can be conceived as an abstract step with only output actions.

### 3.2.2 Some Definitions

In this subsection we provide some definitions that we use in the rest of the thesis.

**Abstract Step Function**

Given a complex state \( u \), \( \text{step}(u) \) is defined as following:

\[
\text{step}(u) = \{ x \in \tau_{\text{abst}}^P | u \in \phi_P(x) \}
\]

**Partitioning Complex Actions**

Given a PIA, \( P \), its set of complex actions \( A_{\text{comp}}^P \) can be partitioned into three disjoint sets, \( \lambda^I_P, \lambda^O_P \) and \( \lambda^H_P \), as defined in following:

\[
\lambda^I_P = \{ a \in A_{\text{comp}}^P | \exists (v, a, u) \in \tau^\text{abst}_P . (\text{InputSchedule}(v, a, u) \neq \emptyset) \}
\]
\[
\lambda^O_P = \{ a \in A_{\text{comp}}^P | \exists (v, a, u) \in \tau^\text{abst}_P . (\text{OutputSchedule}(v, a, u) \neq \emptyset) \}
\]
\[
\lambda^H_P = \{ a \in A_{\text{comp}}^P | \exists (v, a, u) \in \tau^\text{abst}_P . (\text{InternalSchedule}(v, a, u) = \text{sched}(\phi_P(v, a, u))) \}
\]
3.2. PARAMETERIZED INTERFACE AUTOMATA

All Steps for PIA

For a PIA, $P$, the set $\tau_p$ is:

$$\tau_p = \bigcup_{\forall \text{ step } \in \tau_p^{\text{abst}}} \phi_p(\text{step})$$

Moreover, $|\tau_p| = |\tau_p^{\text{abst}}| + |V_p^{\text{comp}}|$.

Size of PIA

The size of a PIA, $P$, is $|P| = |V_p| + |\tau_p|$.

Equivalent IA for PIA

Given a PIA, $P = \langle V_p^{\text{abst}}, V_p^{\text{init}}, V_p^{\text{comp}}, AP^{\text{I,simp}}_p, AP^{\text{Q,simp}}_p, AP^{\text{H,simp}}_p, AP^{\text{comp}}_p, \tau_p^{\text{abst}}, \phi_p \rangle$, $Q = \langle V_p, V_p^{\text{init}}, A_p^{\text{I,simp}}, A_p^{\text{Q,simp}}, A_p^{\text{H,simp}}, \tau_p \rangle$, is the equivalent IA for $P$.

Observation 3.1 Obviously such a $Q$ can be computed in linear time in size of $P$.

Simple Shared Actions

Given two PIAs, $P$ and $Q$ and their equivalent IAs, $P'$ and $Q'$, the set of simple shared actions $\text{SimpShared}(P, Q)$ is defined and is equal to $\text{shared}(P', Q')$.

Complex Shared Actions

Given two PIAs, $P$ and $Q$ and their equivalent IAs, $P'$ and $Q'$, the set of complex shared actions $\text{CompShared}(P, Q)$ is defined as following, if $P'$ and $Q'$ are composable. Intuitively, this set consists of those complex actions that their complex fragments overlap with at least one other complex action’s complex fragment in the other PIA.

$$\text{CompShared}(P, Q) = \{ x \in A_p^{\text{comp}} | (\exists (m, x, n) \in \tau_p^{\text{abst}}) \land (\exists y \in A_q^{\text{comp}} \cdot (p, y, q) \in \tau_q^{\text{abst}})) \land (\exists a \in \text{SimpShared}(P, Q) \cdot a \in \text{sched}(\phi_P(x)) \land a \in \text{sched}(\phi_Q(y))) \}$$

$$\cup$$

$$\{ x \in A_q^{\text{comp}} | (\exists (p, x, q) \in \tau_q^{\text{abst}}) \land (\exists y \in A_p^{\text{comp}} \cdot (m, y, n) \in \tau_p^{\text{abst}})) \land (\exists a \in \text{SimpShared}(P, Q) \cdot a \in \text{sched}(\phi_Q(x)) \land a \in \text{sched}(\phi_P(y))) \}$$

$$\cup$$

$$\{ x \in A_p^{\text{comp}} | (\exists (m, x, n) \in \tau_p^{\text{abst}}) \land (\exists y \in A_q^{\text{comp}} \cdot (p, y, q) \in \tau_q^{\text{abst}})) \land (\exists a \in \text{SimpShared}(P, Q) \cdot a \in \text{sched}(\phi_P(x)) \land a \in \text{sched}(\phi_Q(y))) \}$$

$$\cup$$

$$\{ x \in A_q^{\text{comp}} | (\exists (p, x, q) \in \tau_q^{\text{abst}}) \land (\exists y \in A_p^{\text{comp}} \cdot (m, y, n) \in \tau_p^{\text{abst}})) \land (\exists a \in \text{SimpShared}(P, Q) \cdot a \in \text{sched}(\phi_Q(x)) \land a \in \text{sched}(\phi_P(y))) \}$$
Subfragment Relation

Given two execution fragments or two complex fragments, \( x = (v_0, a_0, v_1, a_1, \ldots, v_n) \) and \( y = (u_0, b_0, u_1, b_1, \ldots, u_m) \), \( x \) is subfragment of \( y \), i.e. \( x \sqsubseteq y \), if \( \text{sched}(x) \sqsubseteq \text{sched}(y) \).

This intuitively means that an execution or complex fragment is subfragment of another execution or complex fragment if the sequence of actions belonging in the first one appears in the second one in the same order with the possibility of extra actions existing in the second one. Checking the subfragment relation can be carried out in time \(|x| + |y|\).

Equality Between Two Complex Actions

Given two PIAs, \( P \) and \( Q \), and two complex actions \( x \in A^\text{comp}_P \) and \( y \in A^\text{comp}_Q \), \( x \) and \( y \) are equal, \( (x = y) \); if \( \phi_P(x) = \phi_Q(y) \).

Embedded Actions Between Two PIAs

Given two PIAs, \( P \) and \( Q \), the set of embedded actions of the two PIAs is defined as:

\[
\text{Embedded}(P, Q) = \{ x \in A^\text{comp}_P \mid (\exists (m, x, n) \in \tau^\text{abst}_P) \land (\exists y \in A^\text{comp}_Q \cdot (p, y, q) \in \tau^\text{abst}_Q) \\
\land \phi_Q(y) \sqsubseteq \phi_P(x) \} \cup \\
\{ x \in A^\text{comp}_Q \mid (\exists (p, x, q) \in \tau^\text{abst}_Q) \land (\exists y \in A^\text{comp}_P \cdot (p, y, q) \in \tau^\text{abst}_P) \\
\land \phi_Q(x) \sqsubseteq \phi_P(y) \}
\]

Intuitively, given two PIAs, \( P \) and \( Q \), their set of embedded actions is the set of complex actions belonging to \( P \), \( Q \) or both. Schedule of each of such complex actions must be subsequence of the schedule of at least one complex action in the other.

Embedding Actions Between Two PIAs

Given two PIAs, \( P \) and \( Q \), the set of embedding actions of the two PIAs is defined as:

\[
\text{Embedding}(P, Q) = \{ x \in A^\text{comp}_P \mid (\exists (m, x, n) \in \tau^\text{abst}_P) \land (\exists y \in A^\text{comp}_Q \cdot (p, y, q) \in \tau^\text{abst}_Q) \\
\land \phi_Q(y) \sqsubseteq \phi_P(x) \} \cup \\
\{ x \in A^\text{comp}_Q \mid (\exists (p, x, q) \in \tau^\text{abst}_Q) \land (\exists y \in A^\text{comp}_P \cdot (m, y, n) \in \tau^\text{abst}_P) \\
\land \phi_P(y) \sqsubseteq \phi_Q(x) \}
\]
Intuitively, given two PIAs, $P$ and $Q$, their set of embedding actions is the set of complex actions belonging to $P$, $Q$ or both. For each of such complex actions, suppose it is $A$, there must exist a complex action, in the other PIA, such that its schedule is subsequence of $\text{sched}(A)$.

### 3.3 PIA Composition

PIA composition is a binary function mapping two composable PIAs into a new PIA. In this section we first define the criteria for composability of two PIAs and then formally introduce the composition. The main difference between PIA and IA composition is in the way complex actions and steps are handled in PIA composition. It is required that after composition of two PIAs, each of complex fragments belonging to the two PIAs, maintain their sequence of actions. Furthermore, it is required that either the whole complex fragment is present in the result or the complex fragment should not appear in the result at all.

#### 3.3.1 Composability

PIA composability follows IA’s composability and, furthermore, introduces extra conditions. The extra conditions require two PIAs to have complex actions that could either be embedded inside each other or do not overlap at all. This is an intuitive restriction since if complex fragments with overlapping actions exist, then composition can not maintain the sequentiality restriction that we have considered for complex fragments. In other words, the complex fragment will be branched in one of its states, which is not desirable.

**Definition 3.9** Given two PIAs, $P$ and $Q$, and their equivalent IAs, $P'$ and $Q'$, $P$ and $Q$ are composable if both of the following are true:

- $P'$ and $Q'$ are two composable IAs.
- $\text{CompShared}(P, Q) = \text{Embedded}(P, Q) \cup \text{Embedding}(P, Q)$. 
The above composability criteria state that two composable PIAs should not only observe the compatibility criteria for IA (criteria in Section 3.1.1) but also should not have any complex actions with overlapping complex fragments. In fact, complex actions should either not have any common actions or one of them should be embedded within the other complex action.

**Observation 3.2** Composability of two PIAs, $P$ and $Q$ can be checked in time $|P| \times |Q|$. This is since the obvious algorithm computes $\text{CompShared}(P, Q)$, $\text{Embedded}(P, Q)$ and $\text{Embedding}(P, Q)$ by simply checking all abstract steps with complex actions in $P$ against similar steps in $Q$.

**Definition 3.10** Given two composable $P$ and $Q$, their sets of interleaved states, $V_{P \# Q}$, and interleaved steps, $\tau_{P \# Q}$, are defined as following:

$$
V_{P \# Q} = V_P \times V_Q
$$

$$
\tau_{P \# Q} = \{((v, u), a, (v', u')) | (v, a, v') \in \tau_P \land a \notin \text{SimpShared}(P, Q) \\
\quad \land v \in V_P^{\text{abst}} \land u \in V_Q^{\text{abst}}\} \cup (1)
$$

$$
\{((v, u), a, (v', u')) | (u, a, u') \in \tau_Q \land a \notin \text{SimpShared}(P, Q) \\
\quad \land v \in V_P^{\text{abst}} \land u \in V_Q^{\text{abst}}\} \cup (2)
$$

$$
\{((v, u), a, (v', u')) | (v, a, v') \in \tau_P \land (u, a, u') \in \tau_Q \land \\
\quad a \in \text{SimpShared}(P, Q)\} \cup (3)
$$

$$
\{((v, u), a, (v', u')) | (v, a, v') \in \tau_P \land a \notin \text{SimpShared}(P, Q) \\
\quad \land v \in V_P^{\text{comp}} \land u \in V_Q^{\text{abst}}\} \cup (4)
$$

$$
\{((v, u), a, (v', u')) | (u, a, u') \in \tau_Q \land a \notin \text{SimpShared}(P, Q) \\
\quad \land v \in V_P^{\text{abst}} \land u \in V_Q^{\text{comp}}\} \cup (5)
$$

State $(v, u)$ is a complex state if either $v$ or $u$ is a complex state. Furthermore, any step $((v, u), a, (v', u'))$ is a complex step if either $(v, u)$ or $(v', u')$ is a complex state.

Maximum number of steps belonging to $\tau_{P \# Q}$ is $(|V_P| \times |\tau_Q|) + (|V_Q| \times |\tau_P|)$. 
Observation 3.3 Observe that $V_{P\#Q}$ and $\tau_{P\#Q}$ can be computed in a straight forward manner in time $|P| \times |Q|$. Also, note that as a result of the product, some of the states in $V_{P\#Q}$ and $\tau_{P\#Q}$ become unreachable from $V_P^{\text{init}} \times V_Q^{\text{init}}$ that can be removed in time $|V_P| \times |V_Q|$. Thus overall time for computing $V_{P\#Q}$ and $\tau_{P\#Q}$ is linear in $|P| \times |Q|$.\footnote{In fact, it can be computed in time $(|P| \times |Q|) - (|\tau_P| \times |\tau_Q|)$.}

Lemma 3.4 For each state $(v,u) \in V_{P\#Q}$, if $(v,u)$ is a complex state then there is not more than one step in $\tau_{P\#Q}$ such that it initiates from $(v,u)$.

Proof. See the proof in Appendix A. □

Lemma 3.5 For each state $(v,u) \in V_{P\#Q}$, if $(v,u)$ is a complex state then, there is not more than one step in $\tau_{P\#Q}$ such that it ends in $(v,u)$.

Proof. See the proof in Appendix A. □

3.3.2 Product

Having understood the properties of the sets of interleaved states and steps, we now provide the necessary formalism for defining the product of two PIAs. Similar to IA, we base the definition of composition for PIA on product definition. Following are necessary definitions, notations and lemmas that enable us to define the product of two PIAs.

Definition 3.11 Given two composable PIAs $P$ and $Q$ and their sets of interleaved steps, $\tau_{P\#Q}$, and interleaved states, $V_{P\#Q}$, the set of illegal complex states of $P$ and $Q$, $\text{IllegalComp}(P,Q)$, is defined as following:

$$\text{IllegalComp}(P,Q) = \{(v,u) \in V_{P\#Q} | ((v \in V_P^{\text{comp}}) \lor (u \in V_Q^{\text{comp}})) \land \left( \nexists ((v,u),a,(v',u')) \in \tau_{P\#Q}\right) \}$$

The set of illegal complex states for two composable PIAs, in fact, identifies those complex states that do not have any steps initiating from them. These states are not appropriate complex states since they can not be part of an abstract step; this is since abstract steps should have complex fragments terminating in an abstract state.
Definition 3.12 Given two composable $P$ and $Q$, and their set of interleaved states $V_{P*Q}$ and steps $\tau_{P*Q}$, the set of legal interleaved states $V_{P\oplus Q}$ and legal interleaved steps $\tau_{P\oplus Q}$ are subsets of interleaved steps and states accordingly that do not contain any illegal complex states.

The algorithm in Figure 3.11 removes illegal complex states in time linear in the size of two composable PIAs. The resulting sets $V_{P\oplus Q}$ and $\tau_{P\oplus Q}$ are used to define the product of two composable PIAs.

Algorithm RemoveIllegalComp($P, Q, \tau_{P*Q}, V_{P*Q}$) :
Variables: $\tau_{P\oplus Q}, V_{P\oplus Q}, L[ ], i, temp$ ;
begin
  $i = 0$ ;
  $L_i = IllegalComp(P, Q)$ ;
  repeat
    $temp = \{(v, u)| \exists ((v, u), a, (v', u')) \in \tau_{P*Q} \cdot (v', u') \in L_i \land$
    $((v \in V_P^{\text{comp}}) \lor (u \in V_Q^{\text{comp}}))\}$ ;
    $L_{i+1} = L_i \cup temp$ ;
  until $L_i == L_{i+1}$ ;
  $V_{P\oplus Q} = V_{P*Q} \setminus L_i$;
  $\tau_{P\oplus Q} = \tau_{P*Q} \setminus \{(v, u, a, (v', u'))| ((v, u) \in L_i) \lor ((v', u') \in L_i)\}$ ;
  return $\tau_{P\oplus Q}, V_{P\oplus Q}$;
end

Figure 3.11: The outline of the algorithm for removing illegal complex states and steps from the set of interleaved steps and states of two composable PIAs, $P$ and $Q$.

Observation 3.6 Computation of $\tau_{P\oplus Q}$ and $V_{P\oplus Q}$ can be carried out in linear time in $|P| \times |Q|$. This is since at most all of the states in $V_{P*Q}$ and steps in $\tau_{P*Q}$ should be checked for illegality which at most requires time linear in $|P| \times |Q|$. 
Lemma 3.7 For each complex state \((v, u) \in V_{P\&Q}\) there exists exactly one sequence \(\langle(v_0, u_0), a_0, (v_1, u_1), \ldots, (v_n, u_n)\rangle\), represented as \(\Delta(v, u)\), such that the following conditions are true:

- \(\exists (v_j, u_j) \cdot ((0 < j < n)) \land (v_j = v) \land (u_j = u)\)
- All \(\langle(v_i, u_i), a_i, (v_{i+1}, u_{i+1})\rangle \in \tau_{P\&Q}\), where \((0 \leq i < n)\), are complex steps
- \((v_0, v_n \in V_P^{abst}) \land (u_0, u_n \in V_Q^{abst})\)

Proof. See the proof in Appendix A. \(\square\)

The above theorem states that for a complex state, \((v, u)\), belonging to the set of legal interleaved states, there exists an execution fragment that \((v, u)\) is part of it and can be assumed as a well-formed complex fragment for an abstract step. In the following theorem we claim that such a complex fragment is actually a complex fragment for an abstract step belonging to \(P\), \(Q\) or both.

Theorem 3.8 Given a complex state \((v, u) \in V_{P\&Q}\) and \(s = \Delta(v, u)\), an alternating sequence of states and actions, only one of the following is true:

- \(\exists x \in \tau_P^{abst} \cdot \text{sched}(\phi_P(x)) = \text{sched}(s)\)
- \(\exists y \in \tau_Q^{abst} \cdot \text{sched}(\phi_Q(y)) = \text{sched}(s)\)
- \(\exists x \in \tau_P^{abst}, \exists y \in \tau_Q^{abst} \cdot (\text{sched}(\phi_P(x)) = \text{sched}(s)) \land (\text{sched}(\phi_Q(y)) = \text{sched}(s)) \land (\text{sched}(\phi_P(x)) = \text{sched}(\phi_Q(y)))\)

Proof. See the proof in Appendix A. \(\square\)

Definition 3.13 Given two composable PIAs \(P\) and \(Q\), and their set of legal interleaved states \(V_{P\&Q}\) and legal interleaved steps \(\tau_{P\&Q}\), we define the sets of first steps of the complex steps, \(\chi_{P\&Q}\), and first states of the complex steps, \(\sigma_{P\&Q}\), as below:

\[
\chi_{P\&Q} = \{\langle(v, u), a, (v', u')\rangle \in \tau_{P\&Q} \mid ((v \in V_P^{abst}) \land (u \in V_Q^{abst})) \land ((v' \in V_P^{comp}) \lor (u' \in V_Q^{comp}))\}\\
\sigma_{P\&Q} = \{(v', u') \in V_{P\&Q} \mid \exists((v, u), a, (v', u')) \in \chi_{P\&Q}\}\\
\]
Observation 3.9 $\chi_{P \otimes Q}$ and $\sigma_{P \otimes Q}$ can be computed in time linear in $|P| \times |Q|$.

Definition 3.14 Given two composable PIAs $P$ and $Q$ and their set of first states of the complex steps, $\sigma_{P \otimes Q}$, and a complex state $(r, s) \in (\sigma_{P \otimes Q})$ where $\Delta(r, s) = ((v_0, u_0), a_0, (v_1, u_1), \ldots, (v_n, u_n))$, we define the injective function $\Psi_{P \otimes Q}$ which maps $(r, s)$ into exactly one step $((v_0, u_0), c, (v_n, u_n))$, where $c \in (A^\text{abst}_P \cup A^\text{abst}_Q)$, as below:

$$
\Psi_{P \otimes Q}(r, s) = \begin{cases} 
((v_0, u_0), a, (v_n, u_n)) & \text{if } (v_0, a, v_n) \in \tau^\text{abst}_P \land \\
& \text{sched}(\phi_P((v_0, a, v_n))) = \text{sched}(\Delta(r, s)) \\
& \land (\#(u_0, b, u_n) \notin \tau^\text{abst}_Q \\
& \text{sched}(\phi_Q((u_0, b, u_n))) = \text{sched}(\phi_P((v_0, a, v_n)))) \\
((v_0, u_0), b, (v_n, u_n)) & \text{if } (u_0, b, u_n) \in \tau^\text{abst}_Q \land \\
& \text{sched}(\phi_Q((u_0, b, u_n))) = \text{sched}(\Delta(r, s)) \\
& \land (\#(v_0, a, v_n) \notin \tau^\text{abst}_P \\
& \text{sched}(\phi_P((v_0, a, v_n))) = \text{sched}(\phi_Q((u_0, b, u_n)))) \\
((v_0, u_0), a, (v_n, u_n)) & \text{if } (v_0, a, v_n) \in \tau^\text{abst}_P \land (u_0, b, u_n) \in \tau^\text{abst}_Q \land \\
& \text{sched}(\phi_P((v_0, a, v_n))) = \text{sched}(\Delta(r, s)) \land \\
& \text{sched}(\phi_P((v_0, a, v_n))) = \text{sched}(\phi_Q((u_0, b, u_n))) \land \\
& (\exists a_m \in \text{sched}(\phi_P((v_0, a, v_n))) \cdot a_m \in \mathcal{A}^L_{P, \text{simp}}) \\
((v_0, u_0), b, (v_n, u_n)) & \text{if } (v_0, a, v_n) \in \tau^\text{abst}_P \land (u_0, b, u_n) \in \tau^\text{abst}_Q \land \\
& \text{sched}(\phi_Q((u_0, b, u_n))) = \text{sched}(\Delta(r, s)) \land \\
& \text{sched}(\phi_Q((u_0, b, u_n))) = \text{sched}(\phi_P((v_0, a, v_n))) \land \\
& (\exists b_m \in \text{sched}(\phi_Q((u_0, b, u_n))) \cdot b_m \in \mathcal{A}^L_{Q, \text{simp}})
\end{cases}
$$
3.3. PIA COMPOSITION

Observation 3.10 $\Psi_{P \otimes Q}$ can be computed in time linear in $|P| \times |Q|$. This is since for each state $(v, u) \in \sigma_{P \otimes Q}$, we have the starting step of $\Delta(v, u)$ in set $\chi_{P \otimes Q}$; by continuing traversing the rest of steps following that step, $\Delta(v, u)$ can be trivially computed. Furthermore, there could only exist maximum $(|V_P| \times |\tau_Q|) + (|V_Q| \times |\tau_P|)$ steps to be traversed to compute all $\Delta$s. Thus the whole computation does not exceed time linear in $|P| \times |Q|$.

Notice that if we have overlapping steps in the product and if the name of the overlapping abstract steps in two PIAs are different then we choose the name of the complex action that has input actions in its schedule. If the names happen to be the same then naturally their shared name is chosen. This choice is motivated by the fact that input complex action can simulate the name of functions or methods in a system and it seems to us such a name will be more valid than the name of the function call, i.e. complex action with output actions.

Product

Having defined the necessary definitions and relations, we are now ready to define the product operation on two composable PIAs. Product of two composable PIAs $P$ and $Q$, is a new PIA presented as $P \otimes Q$, and is defined as following:

\[
\begin{align*}
V^\text{abst}_{P \otimes Q} &= V^\text{abst}_P \times V^\text{abst}_Q \\
V^\text{init}_{P \otimes Q} &= V^\text{init}_P \times V^\text{init}_Q \\
V^\text{comp}_{P \otimes Q} &= \{(v, u) \in (V_P \times V_Q) | (v \in V^\text{comp}_P) \lor (u \in V^\text{comp}_Q)\} \\
A^I_{P \otimes Q} &= (A^I_P \cup A^I_Q) \setminus \text{SimpShared}(P, Q) \\
A^O_{P \otimes Q} &= (A^O_P \cup A^O_Q) \setminus \text{SimpShared}(P, Q) \\
A^H_{P \otimes Q} &= A^H_P \cup A^H_Q \cup \text{SimpShared}(P, Q) \\
A^\text{abst}_{P \otimes Q} &= A^\text{abst}_P \cup A^\text{abst}_Q \\
\tau^\text{abst}_{P \otimes Q} &= \{(v, u), (v', u') \in \tau_{P \otimes Q} | (v, v' \in V^\text{abst}_P) \land (u, u' \in V^\text{abst}_Q)\} \\
\phi_{P \otimes Q}((v, u), a, (v', u')) &= \begin{cases} 
((v, u), a, (v', u')) & \text{if } a \in A^\text{comp}_{P \otimes Q} \\
\Delta(\Psi^{-1}_{P \otimes Q}((v, u), a, (v', u'))) & \text{if } a \in A^\text{simp}_{P \otimes Q}
\end{cases}
\end{align*}
\]
Notice that since $\Psi_{P \otimes Q}$ is an injective function then $\Psi_{P \otimes Q}^{-1}$ is a function.

### 3.3.3 Composition

Similar to IA, for PIA we define composition as a binary function defined on top of product operation. Composition of two composable PIAs results a new PIA. We adhere to concepts such as *illegal states*, *environment* and *legal environment* defined in IA, Section 3.1.3, and propose our composition based on those concepts.

Recall that composition of two IAs is defined by removing their illegal states from their product. Similarly in PIA, we can compute the composition of two PIAs, $P$ and $Q$, $P \parallel Q$, by removing their sets of illegal states, $\text{Illegal}(P, Q)$, as defined in Section 3.1.3. Removing such states could make some other states and steps unreachable.

Notice that by removing illegal states, we may create new “illegal complex states”. This means that we have to run the algorithm in Figure 3.11 twice. However, if we are not interested in computing $P \otimes Q$, we can remove the illegal states earlier, right after computing $V_{P \otimes Q}$ and $\tau_{P \otimes Q}$ and before computing $V_{P \otimes Q}$ and $\tau_{P \otimes Q}$; thus avoiding removing illegal complex states twice. If we adhere to this approach, essentially $P \otimes Q$ would amount to $P \parallel Q$.

We propose our composition algorithm based on the latter approach. In this way, composition of two composable PIAs $P$ and $Q$, $P \parallel Q$, can be computed by carrying out the following steps:

1. Computing $V_{P \otimes Q}$ and $\tau_{P \otimes Q}$
2. Computing $\text{Illegal}(P, Q)$ as described in Section 3.1.3.
3. Computing $V_{P \otimes Q}$ and $\tau_{P \otimes Q}$
4. Computing $\chi_{P \otimes Q}$ and $\sigma_{P \otimes Q}$
5. Computing $\Psi_{P \otimes Q}$ for $\sigma_{P \otimes Q}$
6. Computing $P \otimes Q$
7. Removing unreachable states and steps after steps 1, 2 and 3.
3.3. PIA COMPOSITION

After carrying out these steps, $P \parallel Q$ is computed and is equal to $P \otimes Q$.

**Theorem 3.11** Given two PIAs, $P$ and $Q$, their composability and their composition, $P \parallel Q$, can be computed in time linear in $|P| \times |Q|$

**Proof.** See the proof in Appendix A. □

3.3.4 Discussion

PIAs are intuitive formalisms for modelling software systems that may have composite as well as atomic sets of messages being exchanged. Methods and method calls are examples of composite entities being exchanged among systems. Complex XML data types, which suggest messages with multiple elements, are another application for PIA. As a major example, Web services can use such composite messages as their inputs and outputs.

All of the operations defined on a PIA stays valid for an IA. In fact, each IA could be considered a PIA and conversely we can translate each PIA into an IA, see Subsection 3.2.2.

Recall our argument about IA being space efficient in comparison with I/O automata, PIAs act even more efficiently than IA. PIA composition avoids unnecessary interleaving of states and steps in the resulting PIA. In comparison, PIA composition could be much smaller than the composition of two equivalent IAs. The reason mainly being that PIAs require their abstract steps to maintain the sequence of their constituent complex steps. Any partial sequence is pruned, which could result in substantially smaller automata. This pruning intuitively makes sense; assume that a complex action represents a function, then in analogy, we would like to either have the whole function or none of its constituent parameters.

As an example, Figure 3.12 represents the composition of two PIAs. The composition is carried out on PIA, $CompPay$, represented in Figure 3.10, on page 67, and equivalent PIA of IA $Prod$ in Figure 3.2, on page 57. Compare this composition with the IA composition of two similar IAs in Figure 3.7. Notice that a whole branch of execution becomes invalid in PIA. In fact, many of possible interleavings suggested by IA composition become invalid in PIA composition. PIA can significantly decrease the size of automata. More importantly, PIAs also provide a more precise representation of systems.
3.4 PIA Refinement

The refinement relation we suggest for PIA is quite similar to IA refinement relation. Similarly, a refined interface may have more inputs and less outputs than the interface it refines. In PIA, however, we should also deal with both complex and simple actions.

Recall how IA refinement relation, defined in Section 3.1.4, works on corresponding states of two IAs and checks whether the refinement conditions of having more inputs and less outputs at each refined state is followed. In PIA refinement, we essentially use the same approach but apply it only on abstract states of two PIAs. Furthermore, we also check whether each involved complex action adheres to the refinement conditions of having more input actions and less output actions. To make this definition concrete, we use some notations defined in IA refinement, Section 3.1.4, as well as some other notations, defined below.
3.4. PIA REFINEMENT

Definition 3.15 For each abstract state \( v \in V_P^{\text{abst}} \) of a PIA, \( P \), the set \( \varepsilon\text{-closure}_P(v) \) is defined as the set containing \( v \) itself and the abstract states that can be reached from \( v \) through one of the following:

1. Abstract steps with simple internal actions.

2. Abstract steps with complex actions belonging to \( \lambda_P^H \).\(^{11}\)

Notice that the environment can not realize possible transitions of state \( v \) to any of \( \varepsilon\text{-closure}_P(v) \) and as such it should be able to accept all possible output actions that may happen at \( v \) or \( \varepsilon\text{-closure}_P(v) \). Conversely, the environment can send an input to \( P \), only if \( P \) is capable of receiving that action in \( v \) as well as in all states in \( \varepsilon\text{-closure}_P(v) \).

An appropriate refinement for PIA, \( P \), suppose it is \( Q \), should be able to satisfy all of \( P \)'s behaviors in its different states properly. A proper refinement accepts all inputs of \( P \) and does not issue more output steps than \( P \) does. To make this concept precise in the context of our state-based refinement, we define notations for sets of externally enabled input and output action at each state. However, we partition each of these two sets into two subsets, namely, \( \text{sets of externally enabled simple input/output actions} \) and \( \text{sets of externally enabled complex input/output actions} \). We then base our refinement relation on these notations.

Definition 3.16 For each abstract state \( v \in V_P^{\text{abst}} \) of a PIA, \( P \), the sets of externally enabled simple input and output actions are defined as following:

\[
\begin{align*}
\text{ExtEn}_{P}^{I,\text{simp}}(v) &= \{a \mid \forall u \in \varepsilon\text{-closure}_P(v) \Rightarrow a \in A_P^{I,\text{simp}}(u) \land \exists (u, a, l) \in \tau_P^{\text{abst}}\} \\
\text{ExtEn}_{P}^{O,\text{simp}}(v) &= \{a \mid \exists u \in \varepsilon\text{-closure}_P(v) \cdot a \in A_P^{O,\text{simp}}(u) \land \exists (u, a, l) \in \tau_P^{\text{abst}}\}
\end{align*}
\]

\(^{11}\lambda_P^H\) represents those abstract steps that their complex fragments entirely consists of simple internal actions. \( \lambda_P^I \) and \( \lambda_P^O \), on the other hand, represent those abstract steps that their complex fragments at least consists of one simple input or output action respectively.
Definition 3.17 For each abstract state $v \in V_{abst}^P$ of a PIA, $P$, the sets of externally enabled complex input and output actions are defined as following:

$$
\begin{align*}
\text{ExtEn}_{I P}^{\text{comp}}(v) &= \{ a \in \lambda_I^P \mid \forall u \in \varepsilon\text{-closure}_P(v) \Rightarrow \exists (u, a, l) \in \tau_{abst}^P \} \\
\text{ExtEn}_{O P}^{\text{comp}}(v) &= \{ a \in \lambda_O^P \mid \exists u \in \varepsilon\text{-closure}_P(v) \cdot (u, a, l) \in \tau_{abst}^P \}
\end{align*}
$$

\[\blacksquare\]

Having defined the sets of actions that can be expected from a state $v$ and states reachable from $v$ through internal steps, we now define the sets of states that can be reached from such states and actions.

Definition 3.18 Given a PIA, $P$, an abstract state $v \in V_{abst}^P$ and an action $a \in \text{ExtEn}_{I P}^{\text{simp}}(v) \cup \text{ExtEn}_{O P}^{\text{simp}}(v) \cup \text{ExtEn}_{I P}^{\text{comp}}(v) \cup \text{ExtEn}_{O P}^{\text{comp}}(v)$, the sets of externally targeted states of $v$ by $a$, $\text{ExtDest}_P(v, a)$, is defined as following:

$$
\text{ExtDest}_P(v, a) = \{ u' \mid \exists (u, a, u') \in \tau_P \cdot u \in \varepsilon\text{-closure}_P(v) \}
$$

\[\blacksquare\]

Having defined the necessary notation, we now formally define the alternating simulation between two states belonging to two PIAs. Based on alternating simulation relation, we then define refinement relation between two PIAs.

Definition 3.19 Given two PIAs, $P$ and $Q$, the binary relation alternating simulation $\succeq \subseteq V_{abst}^P \times V_{abst}^Q$ between two states $v \in V_{abst}^P$ and $u \in V_{abst}^Q$ (represented as $v \succeq u$) holds if all of the following conditions are true:

- $\text{ExtEn}_{I P}^{\text{simp}}(v) \subseteq \text{ExtEn}_{I Q}^{\text{simp}}(u)$
- $\text{ExtEn}_{O Q}^{\text{simp}}(u) \subseteq \text{ExtEn}_{I P}^{\text{simp}}(v)$
- $\text{ExtEn}_{I P}^{\text{comp}}(v) \subseteq \text{ExtEn}_{O Q}^{\text{comp}}(u)$
- $\forall a \in \text{ExtEn}_{I P}^{\text{comp}}(v), \forall (m, a, n) \in \tau_{abst}^P \cdot m \in \varepsilon\text{-closure}_P(v) \Rightarrow \exists (p, a, q) \in \tau_{abst}^Q \cdot \text{InputSchedule}(\phi_P(m, a, n)) \subseteq \text{InputSchedule}(\phi_Q(p, a, q))$
3.4. PIA REFINEMENT

- \( \text{ExtEn}_{Q}^{O,\text{comp}}(u) \subseteq \text{ExtEn}_{P}^{O,\text{comp}}(v) \)

- \( \forall a \in \text{ExtEn}_{Q}^{O,\text{comp}}(u), \forall (p, a, q) \in \tau_{Q}^{\text{abst}} \cdot p \in \varepsilon\text{-closure}_{P}(u) \Rightarrow \exists (m, a, n) \in \tau_{P}^{\text{abst}} \cdot \text{OutputSchedule}(\phi_{P}(p, a, q)) \subseteq \text{OutputSchedule}(\phi_{P}(m, a, n)) \)

- \( \forall a \in (\text{ExtEn}_{P}^{I,\text{comp}}(v) \cup \text{ExtEn}_{Q}^{O,\text{comp}}(u) \cup \text{ExtEn}_{P}^{I,\text{simp}}(v) \cup \text{ExtEn}_{Q}^{O,\text{simp}}(u)), \forall u' \in \text{ExtDest}_{Q}(u, a) \Rightarrow \exists v' \in \text{ExtDest}_{P}(v, a) \cdot v' \succeq u' \)

This relation intuitively states that for a given state \( v \in V_{P}^{\text{abst}} \) there is an alternating simulation through state \( u \in V_{Q}^{\text{abst}} \) if \( u \) is receptive to all input actions, either simple or complex, that \( v \) is receptive to. Furthermore, \( v \) does not issue output steps that \( u \) does not. The above definition generalizes this intuition to the case where \( v \) and \( u \) could actually make transition to their neighboring states through internal steps while maintaining the property we are explaining.

**Definition 3.20** PIA \( Q \) refines PIA, \( P, P \succeq Q \), if:

- \( A_{P}^{I,\text{simp}} \subseteq A_{Q}^{I,\text{simp}} \)
- \( A_{Q}^{O,\text{simp}} \subseteq A_{P}^{O,\text{simp}} \)
- \( \lambda_{P}^{I} \subseteq \lambda_{Q}^{I} \)
- \( \lambda_{Q}^{O} \subseteq \lambda_{P}^{O} \)
- \( \exists v \in V_{P}^{\text{init}}, \exists u \in V_{Q}^{\text{init}} \cdot v \succeq u \)

The above refinement definition propagates the alternating simulation relation to all states of the two PIAs. By starting from initial states of two PIAs, the alternating simulation relation would be checked on all corresponding states. As an example, PIA in Figure 3.13, is the refinement of the PIA in Figure 3.10, on page 67. Notice that, this interface is both capable of carrying out payments via credit card number and account number (has
Figure 3.13: PIA, GenCompPay, is the refinement of the PIA in Figure 3.10.

more inputs), however, it does not provide any error number and can only provide reference number as output (has less outputs).

Similar to IA, PIA refinement is compositional.\textsuperscript{12} Consider the composition of Gen CompPay, in Figure 3.13, with PIA equivalent of Prod IA in Figure 3.2. Figure 3.14 represents such a composition; since CompPay $\succeq$ GenCompPay then compositionality property implies that $(\text{CompPay} \parallel \text{Prod}) \succeq (\text{GenCompPay} \parallel \text{Prod})$.

\textsuperscript{12}In this thesis, we do not provide the proof for compositionality property. However, observing that composition on a more refined interface obeys refinement rule of having less outputs and more inputs gives the intuition for such a proof.
Figure 3.14: Composition of two PIAs GenCompPay, Figure 3.13, and the PIA equivalent of Prod, Figure 3.2. This automaton refines the composition in Figure 3.12; it has more input actions and less output actions than the automaton in Figure 3.12, on page 82.
3.5 PIA Properties

Similar to any interface model, PIAs obey interface models’ well-formedness criteria.\(^{13}\) Following are the criteria that we find interesting in stateful interface models:

1. **Commutativity in Composition:** Given two composable PIAs, \(P\) and \(Q\), \(P \parallel Q = Q \parallel P\).

2. **Associativity in Composition:** Given three composable PIAs, \(P\), \(Q\) and \(R\), \((P \parallel Q) \parallel R = P \parallel (Q \parallel R)\).

3. **Compositionality:** Given three PIAs, \(P\), \(Q\) and \(P'\) such that \(P' \succeq P\), \(P\) and \(Q\) are composable and \(P'\) and \(Q\) are composable, then \((P' \parallel Q) \succeq (P \parallel Q)\).

The above three properties can be proven to be correct. In this thesis, we only provide intuition for such proofs.

**Theorem 3.12** Given two composable PIAs \(P\) and \(Q\), \(P \parallel Q = Q \parallel P\).

*Proof Sketch.* See a proof sketch in Appendix A. \(\Box\)

**Theorem 3.13** Given three PIAs \(P\), \(Q\) and \(R\), either some of PIAs are not composable or \((P \parallel Q) \parallel R = P \parallel (Q \parallel R)\).

*Proof Sketch.* See a proof sketch in Appendix A. \(\Box\)

**Theorem 3.14** Given three PIAs \(P\), \(Q\) and \(P'\) such that \(P' \succeq P\), \(P\) and \(Q\) are composable and \(P'\) and \(Q\) are composable too, then it should be true that \((P \parallel Q) \succeq (P' \parallel Q)\).

*Proof Sketch.* See a proof sketch in Appendix A. \(\Box\)

\(^{13}\)In fact, there is no explicit well-formedness criteria mentioned for automata-based interface models in \([dAH01a]\). The well-formedness criteria we suggest are mainly properties which we observed in IA and consider them useful and in accordance with well-formedness criteria of interface theories.
Chapter 4

Case Study: Web Services

In the rest of this thesis, we show how different interface models, introduced so far, can be employed to model Web services. We show how interface models are helpful in modelling different aspects of Web services and how interface models facilitate different operations, such as discovery of Web services and their composition.

This chapter provides the necessary background for understanding the rest of the thesis. We begin with a brief introduction to Web services and related technologies and provide an overview of core related standards and technologies. There are different sets of proprietary technologies developed in the area of Web services, however in this thesis, we are only interested in core technologies relevant to this thesis. Three main technologies in this respect are: WSDL [CGM+04], UDDI [Bea03] and SOAP [GHM+03a].

We also provide an overview of workflow and workflow management systems. Workflows are used to specify the way composite Web services are executed. We will discuss different workflow patterns and show their relevance in the context of Web services technology. In Section 4.7, we will show how PIAs fit the characteristics of workflow-based systems and how workflow patterns can be simulated by PIAs.

4.1 Web Services

“A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format
We consider WSDL, SOAP and UDDI as three main elements of Web services technologies and discuss them in more details in the coming sections.

- “Web Services Description Language (WSDL) provides a model and an XML format for describing Web services. WSDL enables one to separate the description of the abstract functionality offered by a service from concrete details of a service description such as “how” and “where” that functionality is offered. This specification defines a language for describing the abstract functionality of a service as well as a framework for describing the concrete details of a service description.” [CGM04].

- (Simple Object Access Protocol) “SOAP is a lightweight protocol intended for exchanging structured information in a decentralized, distributed environment. It uses XML technologies to define an extensible messaging framework providing a message construct that can be exchanged over a variety of underlying protocols. The framework has been designed to be independent of any particular programming model and other implementation specific semantics” [GHM03a].

- “Universal Description, Discovery and Integration, or UDDI, is the name of a group of web-based registries that expose information about a business or other entity and its technical interfaces (or APIs). These registries are run by multiple operator sites, and can be used by anyone who wants to make information available about one or more businesses or entities, as well as anyone that wants to find that information. There is no charge for using the basic services of these operator sites” [Dra02].

In fact Web services can be considered software systems that are used through their interfaces which are specified by a certain standard, namely WSDL. Web services rely on message passing for invocation of their functionalities. SOAP is an industrial de facto standard which is responsible for carrying messages over underlying network protocols in a uniform way. The message patterns, however, are independent of the underlying network protocols and are exchanged similarly. In other words, the conveying underlying network protocol does not affect the high level format of SOAP messages. SOAP messages will be bound and presented according to their network protocol when necessary.
4.1. WEB SERVICES

UDDI is basically a set of specifications that provide a unified way for creating repository of Web services’ specifications. Such repositories provide the means for advertising and searching for web services. UDDI is a specification being used and supported by some major industrial companies. Figure 4.1 represents a simplified scenario for a typical Web service execution. It is important to note all messages are exchanged in XML format and the service requester and service provider do not have any prior knowledge of each other. This suggests a loosely coupled model.

Web services are different from other distributed models of computation in the sense that the protocols and communication mechanisms are designed to be independent of hardware as well as underlying software platforms. In fact Web service users could be not aware of the platform on which the web service is running on. To access a Web service
functionality, a service requester needs only to know the address of the service, its *URI*, and the interface of the method which is desirable to be invoked. By using *SOAP* messages, requests are sent to the Web service and consequently the response is received through another *SOAP* message.

In the following sections, we provide some more details about each core technology. We avoid providing comprehensive details; the descriptions are only meant to help the readers to understand the rest of the thesis.

4.2 **WSDL**

In this section we describe *WSDL* and its overall design. *WSDL* specification is a joint effort by a few industrial companies for creating a standard Web service description language. W3C introduced *WSDL* as its working draft and it is expected that ultimately it will become an announced standard for describing Web services interfaces. Currently, *WSDL* is almost accepted as a de facto standard in industry; in fact in absence of any other Web service description language, it is being widely used by different parties.

One of main purposes of Web services is to integrate heterogeneous applications that are residing and running on different platforms. In fact, Web services advocate a model of computation where software systems communicate over the internet to achieve a certain task. Currently the web is mainly realized as a medium between human and computer; the new paradigm emphasizes the importance of autonomous application to application interactions. To achieve this vision, it is required that Web services be specified and described in a *machine processable* and unambiguous manner. *WSDL* specification is defined using *XML* grammar, *XML* and *XML Schema* [TBMM01], which means, it is based on an industry standard. Furthermore, *XML* is platform neutral and thus provides means for specifying Web services in an standard and platform neutral manner.

Another key design goal for *WSDL* is clear separation of two types of specifications: *abstract* and *concrete*. Abstract parts of *WSDL* specification specify the functionality of a

---

1. A Uniform Resource Identifier (URI) is a compact string of characters for identifying an abstract or physical resource. [BLFM98]
2. World Wide Web Consortium
3. Extensible Markup Language
Web service in an abstract level not providing any information about how we can execute the service. Abstract elements of WSDL essentially specify the name and type of the messages exchanged between the Web services and the service requesters. Concrete elements, on the other hand, extend the abstract description to cover the physical address of the Web service along with the wire protocols on which the messages should move. In fact the concrete elements describe how and where service requesters can access the Web service.

In conventional component paradigm, it is almost impossible to talk about standard, platform-neutral description of components and their interfaces; let alone clear separation between abstract interface description of components and their implementation specific aspects. Usually components and their interface descriptions are available through complicated vendor-specific formats; and even understanding the format of an interface description, does not necessarily mean straightforward deployment of that component. On the other hand, Web services provide the grounds for developing distributed systems which can be easily specified, discovered and deployed.

Separation of abstract and concrete elements in WSDL provides the opportunity to practice efficient specification of Web services. As an example, a useful scenario is where a Web service provider would like to expose a single Web service through different interfaces, based on different protocols and messaging systems. The service provider could specify the abstract description of the Web service once and introduce multiple concrete specifications. Another useful scenario would be to expose the same service in different physical addresses.

WSDL elements are named in forms of XML Qualified Names [BHL99]. Qualified names consist of a Prefix and a LocalPart section; the Prefix part is a URI that associates the prefix with the definition of an XML Name Space; the Localpart provides local part of the name. As an example text:file and image:file are two qualified name where “text” and “image” are prefix parts of the names and “file” is the local part. Notice that the presence of prefixes enables us to have similar local names, i.e. “file” in this example. Name spaces and qualified names provide a neat modular approach for maintaining set of names and vocabularies in XML. Two qualified names are equal if all their characters are the same.

---

4An XML namespace is a collection of names, identified by a URI reference [BHL99].
Figure 4.2 shows different WSDL elements and their correlations. Boxes containing other boxes represent containment relation while arrows represent reference relations. Each Web service can be identified by the service element; each service in turn contains multiple endpoints which specify the physical address from which the service can be accessed. Each endpoint refers to a binding element that is responsible for specifying the network protocols by which the service can be deployed. Notice the nice separation of abstract and concrete elements. Interfaces, operations and messages are abstract elements which are used to describe a Web service but they are solely used for specifying the logical aspect of the service. Binding elements bridge between the physical and logical parts of a WSDL specification.
Before describing each WSDL element in more details, we present a sample WSDL description of a Web service (only the abstract section of the specification). Figure 4.3 represents an WSDL interface containing an operation. The operation receives an input message and sends an output or fault message to the service requester. Each operation has a message exchange pattern which specifies the scenario and temporal order of its input and output messages. As such, each message has messageReference attribute specifying the role of each message in that operation, according to that operation message exchange pattern. In this example the operation exchange pattern is specified as In-Out.

The content and type of each input and output message can be either described in the same WSDL document or in another document specifying all complex data types for messages. In this example, ‘cp’ specifies the name space in which those data types are defined. We also assume that cp:Payment contains two elements, namely, credit_no and canadian_price, which represent the credit card number from which the payment should be carried out and the amount which should be paid. cp:ref_no and cp:err_no are integers representing a reference and an error number respectively.

4.2.1 Interface

An interface comprises of a set of WSDL operations (Subsection 4.2.2). Each operation in turn consists of a set of messages (Subsection 4.2.3). Interfaces can extend other interfaces; an extended interface should provide all the operations of the interface it is extending plus new operations it may introduce. An interface can extend multiple interfaces as long as name collision does not happen. Polymorphism is not possible since WSDL specification requires that no two elements of the same type have similar names.

4.2.2 Interface Operation

Each interface can contain multiple operations. An operation is defined through its messages. Messages are either in forms of ordinary messages, which provide means to establish communication between the Web service and the service requester, or fault messages. The pattern with which an operation’s messages are exchanged, defines the operation functionality. Different message exchange patterns [GLS04] are defined and operations can be
<interface name="CreditPay">
  <operation name="CreditPay"
    pattern="http://www.w3.org/2003/11/wSDL/in-out">
    <input messageReference="A" message="cp:Payment"/>
  </input>
  <output messageReference="B" message="cp:ref_no"/>
  <outfault name="err_no" messageReference="B"
    message="cp:err_no"/>
</operation>
</interface>

<element name="CreditPay">
  <complexType>
    <sequence>
      <element name="credit_no" type="double"/>
      <element name="Canadian_price" type="double"/>
    </sequence>
  </complexType>
</element>

<element name="ref_no">
  <complexType>
    <all>
      <element name="reference_no" type="double"/>
    </all>
  </complexType>
</element>

<element name="err_no">
  <complexType>
    <all>
      <element name="error_no" type="double"/>
    </all>
  </complexType>
</element>

Figure 4.3: A simplified WSDL specification for a Web service operation. The operation receives information for carrying out a credit card payment and either returns back a reference number or an error number. Input, output and fault messages are further specified by defining necessary data types. Notice that this partial WSDL specification is only aimed at specifying abstract elements of the service.
participant of any of those patterns. *In-Out* [GLS04] pattern is one of the defined patterns and represents typical operations where exactly one message is sent to a Web service and exactly one response message is received. Operations have a message exchange pattern property which represents their model of message exchange. Messages on the other hand should be defined properly to match the exchange pattern. As an example, *In-Out* pattern requires the operation to have exactly two messages: one message with *in* direction and the other one with *out* direction. Table 4.1 provides the list of currently available “message exchange patterns”.

Operations can also send and receive fault messages. It is possible to associate a message to a fault; furthermore, each message exchange pattern inherently obeys a specific fault generation rule [GLS04]. Following are three possible WSDL fault generation rules:

1. **Fault Replaces Message**: This error generation rule replaces an ordinary message with a fault message. The replacement pattern can be applied to all messages except the first message.

2. **Message Triggers Fault**: This rule lets any message to trigger a fault message. The recipient of an ordinary message may send a fault message in reverse direction of the triggering message, i.e. send the fault message to the originator of an ordinary triggering message.

3. **No Faults**: No fault rule may be applied.

Table 4.1 illustrates the fault generation rules for each “message exchange pattern”.

Similar to interfaces, operations do not support polymorphism. That is to say, if we have two operations with the same qualified names belonging to the same interface, they should either be equivalent or an error occurs.

### 4.2.3 Message

Each operation is defined by its message references. Message references should comply with the operation’s “message exchange pattern”. As an example, if an operation is *In-Out* then it should have exactly two message references, one input and one output message.

Each message reference has three key properties:
<table>
<thead>
<tr>
<th>Message Patterns</th>
<th>Description</th>
<th>Fault Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>In-Only</td>
<td>A single message is sent to the Web service.</td>
<td>No Faults</td>
</tr>
<tr>
<td>Robust In-Only</td>
<td>A single message is sent to the Web service and the Web service may send a fault message back to the service requester.</td>
<td>Message Triggers Fault</td>
</tr>
<tr>
<td>In-Out</td>
<td>A message is sent to the Web service and exactly one message is received by the service requester. A fault message may be sent back to the service requester.</td>
<td>Fault Replaces Message</td>
</tr>
<tr>
<td>In-Multi-Out</td>
<td>A message is sent to the Web service and zero or more messages are sent back to the service requester. Any of messages sent back to the service requester could be a fault message.</td>
<td>Fault Replaces Message</td>
</tr>
<tr>
<td>Out-Only</td>
<td>The Web service sends a message to the service requester.</td>
<td>No Faults</td>
</tr>
<tr>
<td>Robust Out-Only</td>
<td>The Web service sends a message to the service requester and the service requester may send a fault message back</td>
<td>Message Triggers Fault</td>
</tr>
<tr>
<td>Out-In</td>
<td>The Web service sends a message to the service requester and the service requester in turn sends back exactly one message back. The service requester may send a fault message back to the Web service.</td>
<td>Fault Replaces Message</td>
</tr>
<tr>
<td>Asynchronous Out-In</td>
<td>The Web service sends a message to the service requester and the service requester optionally sends one message back. Any of the exchanged messages could trigger a fault message at the receiving point.</td>
<td>Message Triggers Fault</td>
</tr>
<tr>
<td>Out-Multi-In</td>
<td>The Web service sends a message to the service requester and the service requester in turn sends back zero or more messages back. The service requester may send fault messages back to the Web service.</td>
<td>Fault Replaces Message</td>
</tr>
</tbody>
</table>

Table 4.1: Different “message exchange patterns” in WSDL. Each pattern obeys a certain fault generation rule.
4.2. WSDL

- **Message Reference**: Based on a certain “message exchange pattern”, there is some role for each message. This property specifies the role of a certain message in an operation. For example if an operation is In-Out then it would have two messages with appropriate “message reference”, i.e. one input and one output.

- **Direction**: It is either in or out. This values should be consistent with “message reference” value otherwise an error would occur.

- **Message**: The real structure of the message is defined by referencing some elements defined in a type system, typically XML and XML Schema elements.

4.2.4 Fault

As mentioned earlier, fault messages are associated with operations’ messages. Furthermore, each “message exchange pattern” adheres to a certain “fault generation rule” and thus the fault messages’ directions and the ordinary messages they represent should be compatible. As an example, if a fault message is associated with an input message and the fault generation rule is “Fault Replaces Message” then the direction of the fault message should be also in.

Each fault message has the following properties:

- **Name**: This is a name by which the fault message can be identified. Furthermore, if multiple operations belonging to the same interface use the same name for their fault messages that would imply that the fault messages are the same and furthermore, the content of the fault messages should be the same.

- **Message Reference**: Based on a certain “message exchange pattern”, there are some role for each message. This property specifies the role of a certain message in an operation. For example if an operation is In-Out then it would have two messages with appropriate “message reference”s, i.e. one input and one output.

- **Direction**: It is either in or out. This value should be consistent with its corresponding “message reference” value otherwise an error would occur.
• **Message**: The real structure of the message is defined by referencing some elements defined in a type system, typically XML and XML Schema elements.

### 4.2.5 Feature

*Features* are WSDL elements that can appear as children of different WSDL elements, e.g. interfaces, operations. Each feature is defined by its *URI*. The *URI* could represent a notion related to a functionality. As an example, by defining a feature for an operation, we can define a certain level of security requirements when dealing with that operation. The semantic of feature is supposed to be specified by its *URI*; in effect the *URI* should be semantically meaningful for all service requesters unambiguously. In other words, once a service requester recognizes a feature, the corresponding *URI* should in fact represent that feature. Features also have an attribute, *required*, which specifies whether the feature is mandatory or optional.

Basically, features provide means for better specification of Web services. In order to use them, it is important that their *URIs* to be defined in an standard way, meaningful for all involving parties.

### 4.2.6 Property

*Properties* are WSDL elements that similar to features can appear as children of some other elements, e.g. interfaces and operations. A property can be conceived as similar to properties of a class or an object. WSDL properties are identified by a *URI* and a *required* property which specifies whether the use of that property is mandatory or optional. The possible values for a property can be specified by using either *constraint* or *value* property. A possible approach for assigning values to aforementioned properties is using XML Schema data types and constraints.

### 4.2.7 Binding

So far we have described different *abstract* elements of WSDL. Once the abstract elements are defined, the abstract elements can be bound to physical concrete communication protocols.
4.3 SOAP

Bound elements provide access points for service requesters to invoke Web services. Each binding mechanism defines its own sets of elements, i.e. binding extension elements [MS03]. As part of current WSDL specification, three sets of binding elements, message formats, are defined: SOAP [GHM+03a], HTTP [FGM+99] and MIME [FB96].

It is possible to define a binding in different levels of a WSDL document, namely, interfaces, operations, messages (both ordinary and fault messages). Once a binding is specified for a WSDL element containing other elements, that binding is valid for all contained elements. As an example, if we define a binding element for an operation then its messages also adhere to that binding specification.

4.2.8 Endpoint

Endpoints are children of WSDL service elements (see 4.2.9). As such they provide the real physical point of access to a specific binding. In fact each endpoint refers to an already defined binding.

4.2.9 Service

A service represents a real world Web service in WSDL. Each service instantiates an already defined interface and comprises of a set of endpoints. In fact through a service element, we can trace back to all other involved WSDL elements of a service.

4.3 SOAP

SOAP is a light-weight message exchange pattern designed for transferring Web service messages over the network. SOAP specification, similar to other Web services core technologies, is XML based and aims at specification of mechanisms for decentralized and distributed message exchange patterns. As opposed to other protocols where they are built on top of binary formats; SOAP is built on top of XML, text messages, and thus provides a suitable ground for interoperability. SOAP is meant to be independent of underlying transportation technologies and as such separates two main concerns in its specification: Message Structure and Transport Protocols. SOAP can be configured to work with different transport
technologies and different bindings. HTTP [FGM+99] is naturally considered a good candidate in this respect. Indeed, SOAP specification is accompanied with binding specification for HTTP [GHM+03b].

SOAP message exchange patterns provide means to define scenarios for SOAP messages. A typical useful pattern is remote procedure call (rpc). SOAP realizes rpc by using some XML structures for representing rpc elements. A method name along with its parameters can be modelled by some struct-like XML elements. The response message would typically contain the output parameters along with possibly a return-value.

SOAP also introduces a specific format for returning fault messages, in a message exchange pattern. It is possible to specify fault-code along with more details about the fault, such as a text description of the fault. SOAP and WSDL resemble each other in different respects; different WSDL message exchange patterns correspond to different SOAP exchange patterns. In fact these two specifications are meant to complement each other. As we mentioned earlier, WSDL documents define a SOAP binding which is responsible for physical message exchange of WSDL operations. WSDL , on the other hand, is mainly responsible for logical description of Web services.

In the following sections we briefly study the format of messages in SOAP and explain how messages are processed and transported. Further details about SOAP is out of the scope of this thesis.

4.3.1 SOAP Message Structure

SOAP messages are enclosed by an envelope tag. A message is comprised of an optional header and a body element. The header element is designed for conveying non-informational parts of the message; usually control information aimed at intermediary nodes $^{5}$ [GHM+03a] are put in header section of the message. Each header element may contain multiple items each targeted at some specific nodes. Headers are meant to be used to specify application-specific controlling mechanisms. Later we would see how we can use the header information to direct different nodes to perform specific tasks. SOAP does not specify how headers should be used and therefore lets application designers to use header elements in different application-specific scenarios.

$^{5}$Nodes between the source and destination nodes that the message travels through.
SOAP body element is the mandatory element of a SOAP message. The body part basically contains the real information that is supposed to be sent from a source to a destination node. The body part is comprised of XML structures properly enveloped to match the receiving node expectations.

Both header and body parts can set the value of their encodingStyle attribute. It is also possible to set the encodingStyle of a certain element of header or body individually. The encodingStyle attribute shows how the enclosed information in an element should be interpreted and serialized. This is an application-specific attribute that lets the designers to use it in different scenarios. For example it is possible to set this attribute to RDF encoding style which is basically a semantic-enabled format for information specification, we will study RDF in Section 5.1.2.

4.3.2 SOAP Messages’ Processing

SOAP messages may travel among different nodes until they reach their destinations. Each intermediary node may do some processing on a certain message and pass it to the next node. Header information of a certain message often specifies which node should do what sort of processing. Normally once a certain header block is processed it is removed; however, often the processing node reinserts the same header block for other interested nodes to process. role attribute of a header block specifies how that header information should be handled. There are three standard roles which rule how nodes would process the header information:

- **none**: A header element with role value set to “none” is not processed by any node. The nodes can inspect the header information though.
- **next**: Any node receiving a header block with its role attribute set to “next” should be able to process that header information.
- **ultimateReceiver**: This value means that the header block is meant to be processed by the ultimate target of the message.

Header blocks have a mustUnderstand attribute too; this attribute can be set to true or false. Setting a header mustUnderstand attribute to true means that those nodes that
are target of that message should be able to understand the semantics of that message and process it properly; if a targeted node fails processing such a message, a fault message must be issued. On the other hand, if a header block `mustUnderstand` attribute is set to `false`, it is not required for the target nodes to process that header information.

Another important header block’s attribute is `relay`. `relay` is also a boolean attribute. Setting `relay` attribute of a message to `true` means that the corresponding should be passed to the next node which might be interested in the content of that header block. A `false` value for this attribute represents the default behavior of a node that is to remove the block which is processed by that node. A typical usage is where the `mustUnderstand` attribute of a message is set to `false` and thus a target node does not have to process a certain header block; setting `relay` to `true` in this scenario guarantees that the message is passed among all intermediary nodes and it has the chance to be processed by a node that is capable of processing it.

4.4 UDDI

The ultimate goal of creating a public Web service is to be discovered and used by potential service requesters. Considering the emergence of thousands or perhaps millions of Web services in the coming years, it is crucial for these services to be registered and published in a standard, searchable manner. Service requesters also need to search and find these Web services and service providers in an efficient and program-understandable way.

So far we have seen how we can call a method in a Web service using its technical description provided by a `WSDL` document. We assumed that the service requesters know the address of their desired Web service and there is no need to search for that service. Not always the scenario is this easy, usually service requesters should search for a specific service and acquire the information for invoking that service (either at design or run time). UDDI specification [Bea03] defines a set of elements and APIs that makes finding of a specific Web service among large number of Web services possible. Universal Description Discovery and Integration, UDDI, defines a set of elements and services, supporting the creation and discovery of information about:

- Web service providers and businesses, their identifiers and taxonomical data.
• Web services, their identifiers and taxonomical data.

• Technical interface descriptions and technical finger prints of Web services and bindings.

Wide acceptance of UDDI by over 300 companies, among them some big companies such as HP, IBM, Microsoft and ORACLE, on one hand, and its underlying technologies, HTTP, XML, XML Schema, SOAP and WSDL, on the other hand, has made UDDI a suitable specification for registry of Web services. Currently UDDI version 3.1 specification is in the process of becoming a W3C standard.

UDDI data structures are XML elements and its APIs can be called through SOAP, this means that UDDI specification is platform-independent. On the other hand, although UDDI is primarily a public registry for Web services, it could also be used as a private registry of Web services. For example a consortium of companies could share a UDDI registry to expose their shared Web services only among themselves. A typical scenario for using UDDI could be in large companies with multiple subsidiary companies; subsidiary companies often are independent but yet interact with the mother company in different respects, including through software systems they develop and use. It would be desirable if all these companies expose their services and the describe them in a standard manner, useful for other parties. UDDI could be helpful in this respect; UDDI along with other key Web service technologies introduce the grounds for real, seamless business to business, \((B2B)\) model of application development. Currently ad hoc formats are being used for representing products and applications in a B2B marketplace. In [OF01], authors have demonstrated the complications of understanding and transforming different sets of formats. With emergence of Web services in B2B markets, UDDI tends to be an appropriate repository for companies to publish and search for Web services of their interest.

UDDI nodes host UDDI data. Using UDDI replication mechanism, nodes can be kept synchronized; a set of synchronized nodes could form a UDDI registry. Respectively different registries may form a group, called UDDI affiliations.

UDDI has four core data structures: businessEntity, businessService, bindingTemplate

\(^6\) (B2B) Electronic commerce between businesses, as opposed to between a consumer and a business (B2C). (definition from free on-line dictionary of computing, http://wombat.doc.ic.ac.uk/foldoc/)
and tModel. UDDI core data structures are identified using *unique identifiers*. UDDI specification introduces human readable unique identifiers (meaningful names rather than arbitrary sequence of characters). Unique identifiers could either be generated by registry or specified by the publishers at the time of publishing. To ensure uniqueness of identifiers between registries in a certain affiliation, an *UBR (Universal Business Registry)* must be considered as the root registry; in this way all keys in all registries are checked to be unique in the context of an affiliation.

The relationships between the core data structures are through either *reference* or *containment*. UDDI is sensitive about containment relationship; a single element can not be contained by more than one element, on the other hand, any number of elements can reference a single element. Figure 4.4 shows the relationship between different UDDI data structures. As an example, an instance of a *businessEntity* contains distinguished sets of elements that can not be shared by other *businessEntity* elements. On the other hand, multiple *businessEntity* elements can reference a single tModel element.

It is often required to define a unique concept that can be referenced uniformly by different UDDI elements. tModel, *technical model*, is capable of expressing a concept or model with specific attributes; other elements can then reference that specific tModel through its key. In Figure 4.4, it is illustrated that bindingTemplate can reference the tModel element; other elements can reference tModels as well, not illustrated in Figure 4.4 though. Elements referencing a specific tModel usually show their compliance with what that specific tModel represents. In the following sections, we briefly study the core elements of UDDI.

### 4.4.1 tModel Structure

Service requesters are usually interested in knowing how a certain Web service works, and what specifications and standards it complies with. Technical model (tModel) introduces a flexible and easy way to define these concepts. Describing a Web service, it is important to provide enough well defined information about the service so that service requesters can find and employ services easily. tModels are means for extending UDDI categorization and identification systems. UDDI is equipped with some predefined identification and categorization systems but different businesses and services introduce new requirements for
businessEntity: Information about the party who publishes information about a service.

businessService: Descriptive information about a particular family of technical services.

bindingTemplate: Technical information about a service entry point and implementation specs.

tModel: Description of specifications for services or value sets. Basis for technical fingerprints.

Figure 4.4: UDDI core data structures and their relationships. businessEntity elements contain businessService entities and each businessEntity in turn contains bindingTemplate elements. The arrow lines represent the reference relationship. Different UDDI elements can reference tModel instances.
categorization and identification of their Web services. They can create their own \texttt{tModels} to specify such information.

Once a \texttt{tModel} is created, it can be identified with a unique key, \textit{tModel Key}. Different entities can reference a \texttt{tModel} key for showing their adherence to a certain standard or categorization and identification system. As an example, we can create a \texttt{tModel}, that holds a link to a \texttt{WSDL} interface document. Any UDDI \texttt{businessService} element referencing that \texttt{tModel} would suggest that it is using the same \texttt{WSDL} document. An interesting scenario would be the case where the referenced \texttt{WSDL} document is representing abstract part of a widely used service which is implemented by different parties. All parties implementing that service can reference the aforementioned \texttt{tModel} letting the service requesters discover their services. As an example, consider the credit card payment service, different service providers can provide the payment service via credit cards. If all such services reference a certain \texttt{tModel} then service requesters searching for that technical model would find all different service providers for the credit card payment service.

\texttt{tModel} applications could be categorized into three main areas:

- \textit{Defining the technical fingerprints:} We could advertise the adherence of a \texttt{businessService} to a specific \texttt{WSDL} document. A \texttt{tModel} entity could be created to reference the address of that \texttt{WSDL} document. All \texttt{businessService} entities that comply with that registered \texttt{WSDL} document could easily reference that \texttt{tModel} key. As an example, consider the interface for credit card payment function \texttt{Pay\_credit(credit\_no, expiration\_date, value)}. Furthermore, assume that such an interface is very commonly used, then we can create a \texttt{tModel} for such an interface and let all other services providing such an interface to reference that \texttt{tModel}.

- \textit{Defining value sets:} A typical usage of \texttt{tModel} is using a \texttt{tModel} to show a certain \texttt{businessEntity} belongs to a specific category of business, according to a specific business taxonomy system. To do this, we can use the \texttt{categoryBag} element of a \texttt{businessEntity} and add a \texttt{tModel} reference to it. We can then assign a value to that reference; this simply means the referencing \texttt{businessEntity} belongs to this category, based on the value we set, according to a certain business taxonomy system. As an example, a \texttt{businessEntity} providing credit card payment can specify that
it belongs to banking industry by referencing a certain \texttt{tModel} that represents the banking industry.

The same application exists for \texttt{identifierBag}, \texttt{address} and \texttt{publisherAssertion} structures. They are all enhanced with the ability to reference \texttt{tModels} that can be used to specify their organizational or identity and various other categorization properties.

- \textbf{Defining a find qualifier:} \texttt{tModels} could be used to configure the functionality of some UDDI APIs. For example a certain \texttt{find} API, such as \texttt{find\_service} could be configured, using appropriate \texttt{tModel} settings, to sort its returning results in ascending or descending orders of publish date. In this way, different \texttt{tModels} specify different sorting mechanisms.

Figure 4.5 represents the \texttt{tModel} structure. The hexagonal symbol represents a sequence of elements. Elements with dashed lines are considered optional elements; elements with “+” sign on their rights are complex elements which are comprised of other elements. An instance of \texttt{tModel} itself can contain references to other \texttt{tModel} instances, through its \texttt{categoryBag} and \texttt{identifierBag}.

\texttt{tModel} is meant to introduce semantics to UDDI documents. By defining a certain concept uniquely and let other documents to reference such a concept, it is possible to unambiguously specify properties of Web services. While \texttt{tModel} is very helpful in this respect, it tends to be rather ad hoc and unexpressive. In Section 5.1, we will see how \textit{Semantic Web} is helpful in this respect.

\subsection*{4.4.2 businessEntity Structure}

Each \texttt{businessEntity} provides information about a business, its nature and physical attributes such as name, address and its affiliation to a certain categorization or identification system. A \texttt{businessEntity} could contain zero or more \texttt{businessServices}. Although in a containment relationship a single element belongs only to one element, \texttt{businessService} element could be contained in more than one \texttt{businessEntity} element by using the \textit{projection} \cite{Bea03} mechanism. A simple application of projection is usage of a certain
Figure 4.5: Structure diagram of tModel [Bea03]. The only mandatory element is the tModel name.
businessService by a chain of related companies and businesses. Projection mechanism can be conceived as a special kind of reference mechanism in the context of a containment relationship.

As illustrated in Figure 4.6, except for “uddi:name” element, other elements are optional. In this way, businesses could easily register themselves using only their names. Once a businessEntity instance is created, it would have a unique key, businessKey, which could be identified by that.

### 4.4.3 businessService Structure

A businessService element is a child of a businessEntity element and describes a Web service in business terms. As we mentioned earlier, a businessService could be used by different business entities through projection mechanisms (see Section 4.4.2). Figure 4.7 depicts the structure of a businessService.

A businessService element is identified by its unique serviceKey attribute and is related to its parent businessElement by a businessKey. If the businessKey attribute of a businessService element is different from the publishing business element then that element is being used through projection mechanism. The name element of businessService is mandatory unless businessService is used through projection mechanism, which means the name is defined in another businessService element.

businessService element contains zero or more bindingTemplate elements that provide the technical description of a Web service.

### 4.4.4 bindingTemplate Structure

bindingTemplate contains technical description, or reference to technical description, of a Web service. Using references to tModels, It could also contain information describing the type of service being offered. Figure 4.8 represents the structure of a bindingTemplate; the switch-like symbol suggests that bindingTemplate either contains accessPoint or hostingRedirector element.

The accessPoint element is an URI, typically a URL, representing the network address of the Web service being described. There are four types of accessPoints:
Figure 4.6: Structure diagram of businessEntity element[Bea03].
Figure 4.7: Structure diagram of `businessService` element [Bea03].
Figure 4.8: Structure diagram of bindingTemplate element [Bea03]. The switch-like symbol suggests that bindingTemplate either contains accessPoint or hostingRedirector element.
4.5 DISCUSSION

- **endPoint:** This kind of access point would contain the physical address of the Web service.

- **bindingTemplate:** This suggests that the real access point is defined in another bindingTemplate element, and this element is pointing to that element. This type enables the publishers to share a certain binding among different Web services.

- **hostingRedirector:** The accessPoint can be accessed by querying another UDDI registry; this type enables the publishers to advertise their services in different locations while keeping the original accessPoint in a single location.

- **WSDL Deployment:** This is probably the most useful and normal way of finding the description of a Web service. The access point would contain the address of the proper WSDL document.

The hostingRedirector element is redundant and its functionality is satisfied by accessPoint element. It is only kept for backward compatibility with previous versions of UDDI.

### 4.5 Discussion

While Web services have drawn a lot of attention they can be considered still an immature paradigm of software development; they only provide basic means for application to application paradigm of computation. Although Web services suggest an extremely portable and loosely-coupled paradigm for computation, they only provides very basic technological elements for such a model. In fact, it is not quite obvious what we can do with such basic primitives and how we can fit existing computer systems into this new model of computation; and more importantly what we gain from such an effort.

There are complications surrounding the Web services technologies. For example, Web services can not compete with performance of traditional client-server systems, i.e. pure RPC mechanisms. Security is also another crucial issue, internet and consequently Web services suggest an inherently open architecture which make the whole infrastructure vulnerable to security problems.
Another major draw-back for using Web services is the lack of well-defined methodologies for development of Web service-based software systems. As an example, any non-trivial software system often requires transactional semantics to carry out its complex business transactions. Web services, on the other hand, inherently propose a loosely-coupled model that is incapable of implementing traditional database concept of transactions. There are proprietary efforts to address these fundamental issues but they could not be taken seriously unless they are adapted vastly and could fit properly in the context of existing sets of standards.

Even though Web service technologies are neither totally original nor flawless, they represent an important evolutionary step for building distributed component-based systems in a standard fashion.

In this thesis, we show how, by formalizing Web services characteristics, we can analyze and reason about different aspects of Web services efficiently.

4.6 Workflows

Workflow technologies have been traditionally used as an appropriate formalism in business analysis areas, specifically in the area of business process modelling and coordination. Recently, however, there has been various applications explored for workflow systems.

Workflows are basically meant to specify the sequence, i.e. control flow of execution, as well as the choice of activities, i.e. routing of activities, to be carried out. Activities are connected through transitions and there could exist more than one set of simultaneous transitions active. In other words, there may exist multiple threads of execution in a workflow. Any set of transitions recurring frequently in workflows represents a workflow pattern. A typical workflow pattern is a sequence, which represents sequential execution of multiple activities. While workflows have been identified and categorized more or less, see [vdAtHKB03]; there are neither any uniform definitions for semantics of workflow patterns and nor any formalism to unambiguously specify their attributes. In fact, in different works, workflows have been realized and implemented slightly differently. In this thesis, we deal with more commonly used workflow patterns that more or less have agreed-

---

7 An elementary activity is an atomic piece of work [vdAtHKB03].
upon definitions. However, it is important to note that different languages, may realize a certain pattern differently. In the absence of any formal definition, slight differences in interpretation are unavoidable.

**Workflow management systems**\(^8\) work on top of some workflows to specify and automate a set of business processes. Recently workflows have been used in various scenarios and levels of computer systems. The main application has been in the area of component-based systems. There has been different efforts to adopt component technologies, such as CORBA, to act as workflow management systems. One of main CORBA’s design goals has been high level of interoperability among diverse component technologies, see [Vin97] for details; this makes the integration of different business processes possible. Different workflow technologies have been built around CORBA to help this integration (For details see [MSKW96, DKM+97]).

Web services, on the other hand, provide an even more unified method for business process integration; and thus, have recently been in the center of workflow research. In fact, the future vision for Web services promises collaborative Web services and businesses. The integration for such collaborative Web services can happen seamlessly compared to all other workflow management systems that are based on ad hoc technologies.

The underlying units of computation in Web services are **WSDL operations** which can simply be conceived by their input and output messages. In fact, existing Web service workflow management standards and models are built on top of WSDL model. Figure 4.9 illustrates the stack of Web service technologies. On top of UDDI there are different sets of Web service orchestration and composition standards, all of them relying on some kinds of workflow concepts.

There have been several attempts by industrial as well as academic communities to introduce a standard for workflow specification of collaborative Web services. The proposed formats are basically specified on top of WSDL and are XML compliant and mainly support only basic workflow patterns. Some prevalent formats such as Microsoft XLANG [Tha01] and IBM WSFL [Ley01], are heavily influenced by some in-home products of corresponding companies. In [vdA03], authors perform a review of key Web services integration models.

\(^8\)“A workflow system is composed of a set of applications and tools that allows for the definition, the creation, and the management of the various activities associated with workflows business processes.” [CBS04]
and compare them in different respects. It seems that they resemble each other in different key aspects. **BPEL4WS** [ACD+03], a Web service workflow model based on **XLANG** and **WSFL** is an effort to combine existing models into a single standard. In Section 5.2, we will discuss a completely different approach, i.e. **OWL-S**, for defining workflow management models that is based on *semantic web* concepts.

In the following subsections, we briefly present some key workflow patterns that are most commonly used in web service workflow models. We only present a subset of identified workflow patterns that are more frequently used in different models. It is important to note that many of more advanced workflow patterns can be constructed by using simpler workflow patterns.

<table>
<thead>
<tr>
<th>Flow Languages (XLANG, WSFL, BPEL4WS and OWL-S)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Finding Web Services (UDDI)</td>
</tr>
<tr>
<td>Describing Web Services (WSDL)</td>
</tr>
<tr>
<td>Invocation Protocol (SOAP)</td>
</tr>
<tr>
<td>Data encoding and Definition (XML)</td>
</tr>
<tr>
<td>Internet Protocols (TCP/IP, HTTP, HTTPS, ...)</td>
</tr>
</tbody>
</table>

Figure 4.9: Stack of Web service technologies. **BPEL4WS** along with other workflow management technologies are built on top of core Web services standards, i.e. **WSDL**, **UDDI** and **SOAP**. **OWL-S** is a semantic model for integration of Web services and is itself built on top of other semantic standards that are not illustrated in this figure.
4.6. WORKFLOWS

4.6.1 Sequence

This pattern represents the execution of a number of activities in sequential manner, one after the other. As an example consider two activities: SearchBookTitle, which searches for existence and the price of a book title, and PayBook, which buys a certain book through credit card. SearchBookTitle and PayBook can be sequentially executed to achieve a successful buy.

4.6.2 Split (Parallel Split)

A point in workflow where multiple activities start executing concurrently. The parallelism can be realized in different manners. It is often the case that different activities are run on different threads of control independently. Alternatively, this pattern is realized by a model where activities are interleaved arbitrarily on a single thread of execution; this is of course less commonly implemented. This workflow does not specify whether and how the split activities are synchronized.

As an example for this pattern, consider two activities: SearchBookTitle and SearchBookISBN. These two activities can be executed in parallel and we do not care about their synchronization. As soon as one of them returns successfully with the price of the book, our task is satisfied.

4.6.3 Split-Join (Synchronization)

This workflow again specifies the parallel execution of a set of activities; however, join conditions and synchronization points, are explicitly specified.

As an example for this workflow pattern, consider two activities SearchBookTitle and CheckCreditCardValid which can be carried out in parallel. However, in order to proceed with the payment both activities should finish successfully, i.e. both the title should exist and the credit card number should be valid for the payment activity to start.
4.6.4 Choice

This workflow specifies the execution of one activity among a bunch of activities. It is also possible to specify execution of more than one activity from a list of activities, i.e. multi-choice.

As an example, a choice can be made whether a search for a book could happen by SearchBookTitle or SearchBookISBN activity.

4.6.5 Loop (Cycle)

This workflow specifies repeated execution of one or more activities in a loop. It is possible to specify an explicit termination condition or implicitly assume that activities at some point exit the loop by themselves. while loops and repeat-until are variations of the loop pattern with explicit termination conditions and termination points.

As an example for this pattern, we can assume the SearchBookTitle activity to be in a loop until it is explicitly terminated to move to the next step of execution, e.g. the next step could be the PayBook activity.

4.7 Modelling Workflows Using PIA

PIAs are capable of simulating many workflow patterns. In this thesis, we only show some of the “basic” workflow patterns [vdAtHKB03] that can be simulated by PIA. Notice that many of more enhanced workflow patterns can be implemented by “basic” workflow patterns, see [vdAtHKB03] for such implementations. In next chapter, we would see how following patterns suffice to simulate the control constructs of a language (OWL-S), which is used for specification of collaborative Web services.

Before showing how workflow patterns can be simulated by PIA, we provide some notations that are different from their counterparts in IA.

Definition 4.1 Given a PIA $P$, a state $u \in V_P^{abst}$ is a terminating state if there does not exist any steps initiating from that state. $V_P^{term}$ the set of terminating states is define as following:

$$V_P^{term} = \{v| (v \in V_P^{abst}) \land (\exists (v, a, u) \in \tau_P)\}.$$
4.7. MODELLING WORKFLOWS USING PIA

Definition 4.2 A state \( u \in V_P^{abst} \) is an iterating state of PIA \( P \) if it is not a terminating state and the destination of all transitions initiating from it, are in the initial state. \( V_P^{\text{loop}} \) the set of iterating states can be defined as following:

\[
V_P^{\text{loop}} = \{ v | (v \notin V_P^{\text{term}}) \land (v \in V_P^{abst}) \land (\forall (v, a, u) \in \tau_P \Rightarrow u \in V_P^{\text{init}}) \}.
\]

Now we are ready to illustrate how workflow patterns can be modelled by PIA. Instead of formally specifying the mappings, which are quite trivial but require lengthy formalisms, we provide English description of the mappings along with some descriptive examples.

- **Sequence**: We simulate the sequence workflow by using “concatenate” operator defined in 3.1.2. Concatenation in PIAs is slightly different from what we introduced in IAs, however, the approach remains essentially the same. Similarly, for concatenation to happen between two PIAs, \( P \) and \( Q \), we require the following to be true:

\[
\text{SimpShared}(P, Q) = (A_P^{\text{L simp}} \cap A_Q^{\text{L simp}}) \cup (A_P^{\text{O simp}} \cap A_Q^{\text{O simp}}) \cup (A_P^{\text{H simp}} \cap A_Q^{\text{H simp}})
\]

(4.1)

By removing “terminating” states of the first automaton in the sequence list and directing all the steps ending in those states to the initial state of the next automaton in the sequence list, we can simulate the “sequence” workflow pattern’s behavior. Furthermore, we should direct all steps initiating from “loop” states of the first automaton in the sequence list to the initial state of the second automaton in the sequence list.

Notice that any of the new steps created or those redirected to new states, could be steps with complex actions. Thus for any such steps, function \( \phi \) should be adjusted appropriately. This adjustment could happen only after the set of complex states are modified properly to contain necessary steps to define step functions for new steps.
• **Choice/If-Then-Else:** “Choice” pattern specifies a state in execution where among different possible execution paths one or more paths are chosen to be executed. “If-then-else”, on the other hand, is a special version of “choice” where between two possible execution paths only one of them is chosen to be executed. The decision about which execution path to be taken is typically made based on a condition variable. As for “if-then-else” pattern, the true or false value for condition variable would determine the execution path.

Similar to “sequence” workflow pattern, the condition in equation 4.1 should hold for PIAs involved in “choice” or “if-then-else” patterns. This guarantees that the result of transforming to these workflow patterns remains a valid PIA.

PIA can simulate these two patterns in terms of representing the branching point in the execution. The concept of condition can not be explicitly modelled by PIA. Usually, conditions can be evaluated in run-time and the run-time environment is capable of inspecting its value. Thus we only show how we can express branch points in PIA.

Similarly, we avoid dealing with conditions in other workflow patterns. It is helpful to imagine that at states where decisions should be made, the corresponding coordinator system, which is responsible for execution and gluing of PIAs, would make such decisions and proceed the execution in appropriate paths.

In this way, the mappings for “choice” and “if-then-else” patterns are quite straightforward. It suffices to merge the initial state of the PIAs that are involved in the “choice” or “if-then-else” patterns. The “choice” operation on two PIAs, defined in Section 3.1.2, simulates such a scenario.

Notice that we assume “choice” always selects one execution path. It is also possible to simulate the general case where more than one execution path should be run, given that all execution scenarios are identified and available a priori at design time.

• **Loop:** A PIA can be transformed into an iterating PIA by removing all terminating states and redirecting each step ending in those terminating states to the initial state. This mapping results an infinite loop. It is also possible to simulate patterns such
as “while” and “repeat-until”. Figure 4.10 provides transformation of the PIA in Figure 3.10 into “while” and “repeat-until” patterns.

- **Split/Split-Join**: “Split” specifies a point in execution where multiple tasks are launched to be executed concurrently. “Split” does not specify any synchronization point for the concurrent tasks. “Split-Join”, on the other hand, specifies the synchronization points of concurrent tasks. It is also possible to specify more complicated scenarios for synchronization of different tasks.

IAs, as well as PIAs, are capable of concurrently executing multiple PIAs through composition relation. Concurrent execution of two PIAs can be specified by simply composing them. Based on how we interpret PIA and input and output actions in PIA, concurrency incurred by composition can be conceived in two different ways:

1. **Actions as functions/methods**: In this scenario, input actions are considered as function bodies, where the function calls are executed or the return locations of function calls. Output actions, on the other hand, are considered function calls or the return values generated by executing some functions.

Observe that composition of two PIAs provides all possible interleavings of actions belonging to the two PIAs. In this way, if we have two input actions one after the other, it simply means that two function calls are happening one after the other asynchronously. The result or return point of those function calls happen later in an arbitrary sequence. Note that by considering all possible interleavings, the composition of two PIAs is receptive to different sequence of input values. If there exists some shared actions, they would appear in all possible states where appropriate input and output actions can be consumed.

Consider the composition in Figure 3.12 (page 82), assume that input actions are function calls and output actions are result of such function calls. Notice how it is possible to call “pay_in_cnd” right after calling “prod_id” or “prod_name”. We do not block for the return value of calls to functions “prod_id” or “prod_name”. Instead, a synchronization would properly happen on “cnd_price”.

In fact PIA composition semantics inherently takes care of synchronization points of two concurrent PIAs. In case explicit synchronization points are re-
Figure 4.10: PIA in Figure 3.10 illustrated in loop. Figure (a) shows an infinite or while loop and Figure (b) represents an example of repeat-until pattern.
4.7. MODELLING WORKFLOWS USING PIA

required, they could be simply enforced by removing those interleavings of actions that violate the explicit synchronization scheme. In other words, composition provides all possibilities but it is always possible to prune the result as desired.

2. Actions as messages: In this case input and output actions are considered as messages. Composition in this context represent all possible interleaving sequence of messages. This resembles what is called interleaved parallel [vdAtHKB03] workflow pattern. However, it is important to note that input messages could concurrently be processed and furthermore output messages could be sent to the environment without blocking for any replies.

In fact, the asynchrony in PIA composition provides the chance for concurrency. However, PIA composition does not rule out the possibility of sequential execution of two PIAs either. Sequential execution paths could be removed but their existences do not hurt the desired concurrency. This is since as long as the environment provides input actions of a certain PIA in an order implying concurrency, that automaton would act as it is instructed and thus sequentiality is avoided. On the other hand, in case the environment provides inputs in an order that does not imply concurrency, the automaton acts as a sequential system.

The explicit synchronization points specified by a “split-join” pattern can be simply created by manually omitting some execution possibilities in a composition and thus enforcing the desired synchronizations. However, not any arbitrary synchronization plan can be enforced. This is since PIA composition inherently synchronizes PIAs at their shared actions and thus implements the obvious synchronization of matching input and output actions. This synchronization is essential to PIAs and can not be overridden.
Chapter 5

Web Services and Interface Models

In previous chapters, we introduced interface models and their attributes. We believe that Web services are typical interface systems that present their functionalities through their “interfaces”\(^1\). We argue that Web services are essentially interface models, in the sense discussed in previous chapters. Consider the well-formedness criteria for interface models mentioned in Section 2.1:

1. Interfaces specify how systems can be used.

2. Interfaces only work in helpful environments and are optimistic; they assume that environment would provide their required inputs and accept their outputs.

These criteria fit the inherent characteristics of Web services. Similar to interface models, Web services specify how they can be used. **WSDL**, described in Section 4.2, specifies the signature and the way different functionalities of a certain Web service can be used.

Furthermore, in Section 4.6, we discussed workflow management systems and their relevance in the context of collaborative Web services, or as they are often called composite Web services. The basic assumption for systems based on workflows is the existence of helpful environment which can enforce semantics of workflow patterns. Such environments provide required inputs at proper execution points. Furthermore, such environments accept

\(^1\)Here we use the term “interface” in the context of Web services terminology.
the generated outputs of systems and use those outputs to properly coordinate the execution of the whole system. It can be observed that Web services, as workflow-based systems, rely on helpful environments for their performance. We believe that interface models can be effectively used for modelling Web services and reasoning about their properties.

In the following sections, we show how Web services behaviors can be simulated by PIAs. In this thesis we choose OWL-S [ea03] as the format for specifying the behavior of composite Web services. OWL-S is a semantically enabled standard that views composite Web services in different levels of abstraction. In Section 5.1, we briefly explain why the semantic Web is important and how it could be helpful to specify the behavior of Web services. In Section 5.2, we discuss components of OWL-S and show how they can be mapped into their equivalent PIA models.

5.1 Semantic Web

The Semantic Web is considered the future vision for the Web. In [BLL01], the authors make a descriptive analogy between current Web and semantic Web: the current Web can be assumed as a giant book while the semantic Web is supposed to be a giant database. In fact, the semantic Web is an approach to present information and meta-data about resources on the Web. In this way, properly specified sets of resources on the Web can be unambiguously specified, searched and retrieved.

The vision is to be able to describe resources on the Web in such a way that all interested parties can easily find, understand and use those resources. In fact, semantic description of resources in such a Web should be machine processable and understandable. It is desired that programs or software agents be able to find resources based on their semantic descriptions and furthermore infer properties about such resources automatically. In this way, it is possible to automate complicated tasks that usually require human decisions and intervention.

There are different challenges in this respect:

- How “unambiguous description” of resources can be specified? (Expressing Meanings)
- How is it possible to trust such descriptions? (Security)
5.1. SEMANTIC WEB

- How can we make things work together in such an environment in a collaborative manner? (Semantically Enabled Systems)

Expressing meaning resembles previous research in knowledge representation. However, the semantic Web is based on an inherently distributed and uncontrollable medium which lacks any notion of central control and any centralized information repository. As such, it is required to use more relaxed and distributed approaches for specifying meaning. On the other hand, it is necessary to have a mutual set of vocabularies for exchanging meaning, concepts and resources among interested parties. Without such sets of vocabularies it is not feasible to have real collaboration among applications over the semantic Web. Again, it is crucial to bear in mind that any approach based on existence of central entities of information or coordination would fail in the context of the Web. Thus, there should exist mechanisms to define different sets of vocabularies and, furthermore, interrelate such sets of concepts.

Current semantic-based specification languages, such as RDF [Bec04], RDF-S [Bri04], and OWL [MvH04], are XML-based languages capable of specifying meaning in a distributed manner, i.e., different components of the specification could reside in different logical or physical locations over the Web. While these languages have a certain level of expressivity power, they can not be useful unless different parties can understand their vocabularies, i.e., the terms and tags they use to describe specific meanings.

Ontologies [BLL01] are sets of vocabularies, objects and concepts organized in a taxonomic manner. It is possible to specify the relations and association among different components of a certain ontology. Furthermore, it is possible to have references and relations among different ontologies. This approach seems to comply with Web’s inherent distributed nature.

For “collaborative semantically enabled applications” to work with each other, it is required to have means for validating the correctness and authenticity of resources and provide descriptions about such resources. Notice that resources in the semantic Web are not necessarily the conventional documents that we are dealing with in the current Web. One can imagine annotating and describing network devices as well as abstract concepts²

---

²In fact anything with a URI.
in a semantic manner and letting interested parties either use those resources or use the
descriptions and their related ontologies to express new meanings.

For checking the validity of semantic description of Web resources, mechanisms such as
digital signatures can be used. However, there are still not well-defined methodologies for
validation and security issues of semantic Web.

### 5.1.1 Semantic Web Standards

There are some languages for semantic specification of Web, all of them built on top of XML.
RDF along with RDF-S are considered core languages for creating semantic descriptions. In
fact, some prominent semantic standards, DAML+OIL [CvHH+01] and OWL[MvH04], are
built on top of RDF and RDF-S. OWL is the revision of DAML+OIL language and DAML+OIL
itself is a more enhanced version of DAML.

We briefly study RDF and RDF-S, in Subsection 5.1.2, and then describe the OWL and its
capabilities for expressing meanings, in Subsection 5.1.3. Lastly, in Section 5.2, we describe
OWL-S which is an ontology for semantic specification of Web services written in OWL.

#### 5.1.2 RDF and RDF-S

*Resource Description Framework* (RDF) is an XML-based language for representing inform-
ation about resources on the Web. RDF can be used to annotate and describe any Web
resource that can be accessed via a URI. The RDF model can be conceived as set of tuples
in forms of *subjects*, *predicates* and *objects*. Subject is the resource that is being described,
the predicate is attribute, characteristic or fact that is being specified about that subject
and the object is the value for such an attribute. This model can be compared with as-
sertions in logic. As an example, the plain English sentence: “Canada’s Population is
31,873,880”\(^3\) can be expressed in RDF, by setting “Canada” as the subject, “population”
as predicate and “31,873,880” as object. Of course, It is possible to state multiple facts
about a specific subject. Furthermore, the object part can be a resource itself which can
be further described as an RDF resource.

\(^3\)The information from Canada’s population clock, http://www.statcan.gc.ca/english/edu/clock/
population.htm, on 2004/05/28 at 12:10 pm.
5.1. SEMANTIC WEB

It is also possible to describe groups of resources. *Bag* and *Seq* keywords in RDF can be used to aggregate a set of things. While these constructs specify a group of resources, they do not limit their members. Sometimes it is desirable to precisely specify the members of a group; in these cases the keyword *collection* can be used.

The concept of *choice* can be specified by the *Alt* keyword. As an example, we can translate the English sentence, “Person is either male or female”, to an RDF expression by using *Alt*.

Another interesting feature is the ability to describe an RDF statement itself. In other words, we can specify more description about an RDF statement and further annotate different components of an existing RDF expression. This is called *reification*. Typical use cases of this feature would be for specification of the author or time of an already stated RDF expression.

Using RDF syntax it is possible to somehow describe different resources; however, these descriptions will be rather ad hoc and not useful for public domain. Semantic Web is essentially interested in creating ontologies of meanings that can be shared among different parties in an understandable manner. To achieve such a vision, there should exist a set of common vocabularies that different parties can agree upon their meanings. Obviously, RDF lacks expressivity capabilities to create ontologies. One of the problems in this regard, is the fact that RDF is not capable of categorizing resources. RDF-S, RDF Schema, on the other hand, is designed for creating such vocabularies.

*Classes* and *properties* are the main features that RDF-S adds to RDF capabilities. Classes are meant for categorization of concepts and resources. Each class can be associated with multiple properties. Each class represents a type and, as such, instances of that class can be created. Furthermore, different classes can be related to each other through the *subclass* relation. An instance of a subclass is also a member of the super class. The subclass relation is *transitive* and implicit subclass relations can be inferred from a set of existing subclass relations.

The way classes and properties are presented in RDF-S somehow resembles programming languages class concepts. However, they tend to be slightly different in dealing with properties. Properties in RDF-S are not bound to a certain class. In fact, they are not limited to the scope of a class; instead, they are valid in the scope of a document. Properties
are defined by their *domains* and *ranges*. The domain of a property is the set of classes that can use that property. A property can have multiple domains and/or ranges. Multiple domains would mean that only classes that are subclasses of all the domain classes can have that property associated with them. The range of a property, on the other hand, specifies possible values for a certain property. Similarly, multiple ranges could exist for a single property and they are interpreted similar to multiple domains.

Also, it is possible for a property to be subproperty of another property. The subproperty could be simply associated with the classes that the “property” is associated with.

Note that while it is possible to express different concepts and meanings by classes and properties and relation among them, RDF does not enforce classes to have values for their properties. It depends on the specifier of an RDF document to observe these issues. This is another major difference between conventional programming languages and RDF in interpreting classes and properties. This is due to the fact that completeness is relative in RDF context. The correctness of an RDF document would essentially depend on the expectations that interested interpreting parties have from that document.

Nevertheless, RDF is neither expressive nor precise enough for describing complicated resources and their interrelations. Some richer languages, such as OWL, are built on top of RDF; they are more expressive and capable of imposing the necessary constraints for describing resources.

### 5.1.3 OWL

OWL is a language built on top of RDF and RDF-S, with a richer syntax for specifying ontologies and their relations. OWL is presented in three flavors:

- **OWL Lite**: which is capable of defining classes and properties as well as specifying cardinalities of 0 or 1 for classes.

- **OWL DL**: provides different classification operations, as well as, set operators such as intersection, complement and unions. OWL DL is guaranteed to have computational completeness properties.

- **OWL FULL**: provides an expressive set of syntaxes without any guarantees for computation completeness.
5.1. SEMANTIC WEB

In the context of this thesis, we only consider features in OWL DL. OWL-S, which is an ontology for specifying Web services behaviors in a semantic manner, is built on top of OWL DL’s capabilities. Note that, as expressiveness improves, the compatibility is also preserved. For example, any OWL Lite ontology is a valid OWL DL ontology and any OWL DL ontology is a valid OWL FULL ontology.

Each OWL ontology has an ultimate root which is the “thing” class. Classes in OWL can be both referenced and used as they are or can be referenced and be augmented for specific use cases. The latter approach allows the development of an ontology in a distributed manner. Similar to RDF, classes and properties can be defined and interrelated through subclass and subproperty elements. Unlike RDF, it is possible to specify cardinality constraints for a property that is associated with a class. However, that constraint will not apply to other classes that may use that property. Specifying quantitative constraints on classes and their properties is a very expressive and useful feature. Consider the class “Book” and property “author”, then we would like each instance of class “Book” to have at least one “author”; this is not possible in RDF but possible to specify in OWL.

Another useful feature of OWL is its capability to use XML Schema [TBMM01] data types, which provides commonly used data types. Properties can be bound to those data types. Furthermore, it is possible to specify characteristics of properties. Below, we describe some of those characteristics:

- TransitiveProperty: Properties such as “ispartof” can be defined transitive. If defined as transitive then, if \( a \) “ispartof” \( b \) and \( b \) “ispartof” of \( c \) then since “ispartof” is transitive: \( a \) “ispartof” \( c \).

- SymmetricProperty: Properties such as “sibling” can be defined symmetric if \( a \) is “sibling” of \( b \) then \( b \) is also “sibling” of \( a \).

- FunctionalProperty: This characteristic specifies that a certain property acts as a function for a certain class. As an example, “age” can be considered a functional property.

- inverseOf: It is possible to specify that a certain property \( p1 \) is inverse of property
\[ p_2. \text{ Then the following is true:} \]
\[ p_1(x, y) \Leftrightarrow p_2(y, x) \]

- **InverseFunctionalProperty**: A property \( p \) is “InverseFunctionalProperty” if: \[ p(y, x) \land p(z, x) \Rightarrow y = z \]

By specifying a restriction, it is applied to all classes that a property belongs to. It is also possible to specify restrictions on a property when it is associated with a certain class but not on other classes that can use that property.

A useful feature for bridging between the concepts residing in different ontologies is the capability to state equivalency of two classes or properties. “equivalentClass” and “equivalentProperty” tags can be used to express these equivalences. Conversely, it is possible to express that a class is different from some other classes by using “disjointWith”. This would imply that instances of such a class can not belong to other classes that are announced disjoint.

Also, it is possible to specify that two instances of classes or properties are equivalent; “sameAs” is used to specify that concept. Conversely, it is possible to state that two instances are distinct, by “differentFrom”; or even to state that more than two instances are distinct, using “AllDifferent”.

Some more enhanced features are also available in OWL DL. Set-like operations, such as intersection, union and complement, are available in OWL DL. These operations can be used to define complex classes. As an example, a class can be defined as intersection of two or more classes. Class “mother” can be defined as intersection of classes “parent” and “female”.

OWL provides an expressive set of features which can be used to describe complex concepts. In the next section, we discuss OWL-S, which is an ontology written in OWL to describe Web services semantically.

### 5.2 OWL-S

Web services are expected to play an important role in the future Web. Similar to other Web resources, Web services can be also annotated and specified by semantic Web lan-
5.2. OWL-S

尤其是，在大量Web服务存在的情况下，语义Web可以有效地用于发现Web服务。要语义化地描述它们，需要使用一些足够表达能力和能够描述Web服务不同方面的语义。拥有这样的语义，其词汇和语法可以用来描述Web服务的不同方面。

语义描述Web服务，在某种意义上，可以被认为比其他Web资源更重要。Web服务的描述是为机器而不是人类设计的。虽然一个Web服务可以有一个GUI用于与最终用户交互，但Web服务的愿景本质上是基于在Web上进行分布式计算的想法。因此，Web服务可以有效地利用语义Web概念。

OWL-S是创建这样的语义网络的一个努力。如图5.1所示，OWL-S是建立在OWL、RDF和RDF-S之上的。而WSDL这样的语言是用于指定Web服务的签名和语法，而OWL-S是用于指定Web服务的语义描述。

5.2.1 OWL-S Classes

OWL-S引入了三个主要类来描述Web服务，即profile、process和grounding。注意，不同类之间可能存在不一致性。例如，Web服务的“profile”描述可能指定一些输入和输出元素；而“process”描述则指定一组不同的输入和输出。对于这种特殊情况，the “process” specification overrides the “profile” specification.
• **profile**: The profile ontology introduces necessary classes and attributes to annotate a certain web service in terms of what it does. In fact, service requesters are able to browse through profile classes and search for their desired services. Properties such as “contactInformation”, “QualityRating” and “serviceCategory” are used to present general business-related information about a web service. There are also some attributes such as input, output, precondition and effect for specifying the functionality of the service. More precise specification of functionality can be found in the process class. However, the brief functional description of services in profile class enables service requesters to search for their desired Web services more effectively.

• **process**: Process class uses the concept of process to model Web services. There are three types of process classes, namely, atomic, simple and composite. Atomic processes are meant to model WSDL operations, while composite processes represent sets of collaborating processes that are glued to each other by some control constructs. In Section 5.2.1, we will see more about processes.

• **grounding**: While process class describes the functionalities of a certain Web service, it does not specify how such processes can be reached and executed. Using the grounding class, it is possible to specify the equivalent WSDL document for an atomic OWL-S process specification. Note that, not always, simple mapping between a OWL-S specification and a WSDL description happens. As an example, it might be the case that two message parts in OWL-S will be mapped into one message in WSDL.

In this thesis, we are more interested in studying OWL-S “process” model. The reason being that “profile” and “grounding” classes are helpful for searching and executing composite Web services respectively. In this thesis, however, we are more interested in searching Web services based on their functional attributes, which are mainly found in “process” class. We are also interested in composing Web services from existing Web services, which can again be accomplished by using information in “process” classes.

---

4 Control constructs in OWL-S can be conceived as workflow patterns.
5.2. OWL-S

Process Class

In OWL-S, processes are categorized into three types. Atomic processes represent different kinds of WSDL operations, e.g., In-Only, In-Out. Atomic processes can be invoked directly and only has a single step of execution. The type of the process and the invocation related information can be found in the corresponding grounding document. Composite processes, on the other hand, are sets of atomic or composite processes that are glued together using control structures, e.g. loop, split, if-then-else. These control structures specify the flow of execution between several processes. Simple processes can be used to represent atomic or composite processes in a higher level of abstraction.

Simple processes can provide an abstract view of an atomic or composite process. However, they can not be invoked and thus do not have any grounding class associated with them. They are especially useful for modelling composite processes in different levels of abstraction. A composite process represented by a simple process only models its inputs, outputs, preconditions and effects; this is called black-box view\(^5\) of that composite process. On the other hand, a complete specification of a composite process, in terms of its building atomic processes and control structures, is called its glass-box view\(^6\).

Each process, regardless of its type, can have some or all of these properties:

- **Input**: input messages in OWL-S are similar to WSDL messages and often are mapped to such messages in the grounding class.

- **Output**: output messages in OWL-S are similar to WSDL messages and often are mapped to such messages in the grounding class. Note that output messages could be conditional.

- **ConditionalOutput**: The outputs will be generated by the process only if their associated condition(s) are satisfied. Neither OWL-S nor OWL provide any means to specify these conditions.\(^7\)

- **Precondition**: Preconditions are conditions which should be provided for a process to perform. Again, currently there is no syntax for specifying such conditions.

\(^5\)Compare with stateless interfaces described in interface theories.

\(^6\)Compare with stateful interfaces described in interface theories.

\(^7\)Obviously some kind of logic is required to specify and reason about these conditions.
• **Effect:** After a process execution, some effects are generated in the environment in which that process is executed.

• **ConditionalEffect:** These kinds of effects would happen only if necessary condition(s) are satisfied.

Note that *OWL-S* does not impose any cardinality constraints on the number of instances of a certain property for a certain process. Furthermore, messages could be associated with well-defined entities in *RDF/RDS-S* ontologies and as such those messages could represent semantically meaningful concepts.

**Control Constructs**

Composite processes can be decomposed into their building processes and control constructs. Whenever necessary, the ordering of the processes involved in a certain composition can also be expressed, by using control constructs such as *sequence*. Following is the list of *OWL-S* control constructs:

• **Sequence:** This control construct specifies the sequential execution of a number of processes.

• **Split:** Concurrent execution of a number of processes without specifying their synchronization can be specified with this control construct.

• **Split+Join:** This control construct is similar to *split* plus the capability of specifying synchronization points among some or all of involved processes.

• **Unordered:** No order of execution is specified by this control construct. However, all of the involved processes should terminate their execution.

• **Choice:** This construct specifies the choice of execution from a list of processes. Additional properties are used to specify how this choice should happen.

• **If-Then-Else:** Based on the *condition*, if *true* the *then* process and if *false* the *else* process will be executed.
• **Iterate**: Synonym *Repeat*. This control construct repeatedly executes a process. The termination condition can be specified by either *Repeat-Until* or *Repeat-While*, with similar semantics as in programming language’s *repeat-until* and *while*.

### 5.2.2 Mapping OWL-S Processes to PIA

Composite processes can model a set of collaborative simple processes and their interactions through control constructs. Constituent processes of a composite process could themselves be composite processes. In this work, we are only interested in mapping composite processes that their constituent processes are atomic processes. This is due to the fact that PIAs are stateful interfaces and as such are meant to model stateful interfaces with explicit states of executions. Being able only to map such composite processes is not really restrictive, any composite process, in a semantically well-formed OWL-S document, can be eventually expanded into its constituent atomic processes.

We start by modelling atomic processes. As described in Section 5.2.1, atomic processes can have input and output elements as well as preconditions and effects. Conditions in OWL-S, as well as in OWL, are not specified precisely; authors have postponed the precise specification to future versions of OWL and OWL-S. In this thesis, we are not interested in mapping conditions. As future work we have some concrete plans to expand our work in that direction, see Section 8.2 for details about future work.

**Mapping Atomic Processes**

An atomic process can be simply mapped to a PIA. The name of the PIA is the name of the corresponding process and its messages can be mapped as input and output actions, including conditional output messages being mapped as output actions. The states and temporal sequence of actions in the resulting automaton can be realized by inspecting the grounding class of the atomic process. 8 OWL-S guarantees all atomic processes have grounding documents.

---

8Current version of OWL-S relies on WSDL 1.1 for specification of groundings. In this thesis, we rely on WSDL 2.0 which provides a different scheme for specifying the type of interface operations. See Section 4.2.2. We assume that OWL-S would eventually support current WSDL 2.0 taxonomy for operations and thus propose our mappings based on WSDL 2.0.
As an example, an **In-Out** WSDL operation explicitly declares that an input action, corresponding to **OWL-S** input message, should initiate from the initial state of corresponding automaton followed by an output action, corresponding to **OWL-S** output message. Conditional output messages are dealt similarly to output messages.

Inspecting the WSDL documents associated with **OWL-S** processes, some input or output messages may have complex types, i.e. consisting of different parts. Such message can be mapped to PIAs’ complex actions and the necessary steps and complex action mappings could be added to the $\phi$ and $\tau$ functions of the corresponding PIA.

Fault messages, associated to a certain **WSDL** message, can also be simulated by PIA. Depending on the direction and *fault generation rules* of a certain fault message (see Section 4.2.2 for more details), it is possible to create an appropriate message in the corresponding PIA. As an example, consider an **atomic** **OWL-S** process with operation type **In-Out** where the output message has **Fault Replaces Message** fault generation rule. Such a process could be mapped to a PIA with three states and three steps. The initial state would initiate an input message and the second state would initiate two output messages, one for the output message and one for the fault message, both ending in the third state. Note that only one of the output or fault message will be triggered. Similarly, other **WSDL** operation types and *fault generation rules* can be simulated by PIA.

Figure 5.2 illustrates PIA mapping of an **OWL-S** atomic process in accompanied example in **OWL-S** specification [ea03]. Note that the output message “FAULT” is an imaginary fault message not present in the original specification of the process. “FAULT” is a fault message for the output message “locateBookOutput” which we assume has the fault generation rule of **Fault Replaces Message**.

**Mapping Composite Processes**

A **composite** process can be realized by its constituent **atomic** processes and control constructs. By mapping **atomic** processes to their equivalent PIAs and then gluing the resulting PIAs to each other through the corresponding control constructs, we can map a **composite** process to its equivalent PIA. Control constructs are subset of workflow patterns

---

9For the complete **OWL-S** specification of the example refer to http://www.daml.org/services/owl-s/1.0/examples.html.
and as such they can be mapped to PIAs as shown in Section 4.7.

OWL-S class, ValueOf, and OWL-S property, SameValues, are used to specify the flow of messages within composite processes. In simple processes, the target process for messages are not clear. Using ValueOf and SameValues, the target processes of messages can be specified. A more interesting application is specification of how output messages of a process can be used by input messages in other processes. By specifying these, communication channels between different processes are defined.

The notion of output messages being consumed by input messages of other processes is analogous with the concept of internal actions in PIAs. We map these pairs of messages as internal actions. When the names of input-output messages are not similar, we tend to adopt the input message name as the new internal action name.

The mapping can be carried out by considering the highest level specification of a composite process and identifying its constituent processes, they are either atomic or composite processes themselves. Each of such constituent composite processes is in turn inspected for its constituent elements recursively. Such inspections result in glass-box view of that composite process that can be mapped into their equivalent PIAs. By mapping the pro-
cesses in a bottom-up fashion and assembling the resulting PIAs by their gluing control constructs the complete mapping of the composite process can be generated. Note that PIA is capable of simulating different workflow patterns, i.e. OWL-S control constructs, and thus, a complete mapping of a composite process can be produced.

Figure 5.3 illustrates the PIA mapping for a book store service specified in OWL-S. This figure is the mapping for the highest level composite process which coordinates all involved atomic processes.\(^\text{10}\) In Figure 5.4, we have provided some parts of the OWL-S specification for this book store system.\(^\text{11}\) The system starts by receiving a book name as input; if the book is available then the user can login to the system, or create an account if there is a new user, and by specifying the payment and delivery arrangements the book can be purchased and delivered.

Observe in Figure 5.3 that except for a “choice” control construct, “choice” in state “3”, the rest of the processes are linked to each other through “sequence” control construct. Step (2, putInCartBookISBN, 3) is an internal step and is realized by inspecting the dataflow document of the bookstore process model. Also, notice that (3, createAcctInfo, 5) and (5, createAcctOutput, 7) are steps with complex actions and each are comprised of two complex steps. Any message consisting of different elements can be conceived as complex actions and its constituent elements can be considered simple actions.

Figure 5.4 illustrates some parts of the OWL-S specification of this book store system. The specification can be mapped to a PIA, as done in Figure 5.3. The complete OWL-S specification is a long document specifying in details all processes (Atomic, Simple and Composite) and all involved messages. There also exists a data flow document which specifies the correlations of messages. The mapping that we have provided in Figure 5.3 is the mapping for the highest level composite process, i.e., FullCognitoBuy process. However, as mentioned earlier, the mapping consists only of Atomic processes. In fact, we have traced all of the Composite and Simple processes back to their constituent Atomic processes and their involved control constructs.

\(^\text{10}\)Figure 3.1 is IA mapping of the same process which does not show the constituent simple actions of the complex actions, as shown in Figure 5.3.

\(^\text{11}\)For complete specification of this service refer to: http://www.daml.org/services/owl-s/1.0/examples.html
Figure 5.3: Presentation of a OWL-S composite process specifying an online book store. This composite process consists of two complex actions and an internal step.
Figure 5.4: Some parts of the OWL-S specification for composite process FullCognitoBuy. FullCognitoBuy is the sequence of two other processes, i.e., CognitoBuyBook and LocateBook. CognitoBuyBook is itself another composite process and LocateBook is an atomic process with a conditional output. LocateBook process receives a "BookName" and in case the book is available sends information about the book, i.e., LocatedBookOutput, as the output of the process.
Chapter 6

Web Services Discovery

Web services are meant to be discovered and used by different interested parties over the Web. As such, interested parties should be able to search for and evaluate possibly useful Web services that are capable of satisfying their requirements. In the presence of a large number of Web services it is difficult to manually evaluate those artifacts. In fact, it is expected that in the coming years a large number of public Web services will become available over the Web. Thus, it is important to address this issue in a formal and efficient way.

Searching for Web services consists of two main activities:

- Searching for Web services based on some non-functional requirements, such as quality of service and cost.

- Searching for Web services satisfying a certain functional requirement, e.g., a Web service for credit card payments.

There has been some efforts to model non-functional attributes of Web services and performing searches over such Web services. In [ZBD+03], the authors propose an approach for performing searches for Web services based on qualities such as “reliability”, “cost”, “quality of service” and etc. Currently, there is no well-defined language/specification model for specifying non-functional attributes of Web services.

As for searches based on functional requirements, similar to authors in [ZW96, ZW97], we believe that software matching activities, based on functional requirements, could be
categorized into two main concerns, namely, signature/non-behavioral matching [ZW96] and specification/behavioral matching [ZW97]. The former mainly investigates interfaces, functions, methods, modules and their parameters for comparison while the latter investigates the preconditions and effects of software artifacts and compares them against a certain user requirement/request. Following is a brief description of the types of tasks in each area in the context of Web services:

- **Signature/Non-Behavioral Matching:** Web services are based on a message-passing paradigm where messages could represent information items, function calls and etc. As such, similarity among messages of two Web services (or between a query and an a certain Web service) could represent some potential matching. Furthermore, comparison of messages may not necessarily be limited to exact syntactical comparisons. Semantic web techniques could be used to make this process more efficient and practical. Moreover, by specifying the temporal sequence of messages and dependencies among input and output messages of queries and comparing these information against individual Web services, it is possible to find more precise matches.

- **Specification/Behavioral Matching:** While similarity between signatures and messages of a Web service and a query may imply good chance for their functional similarity, it can not guarantee a certain match. To guarantee a match, it is required for a Web service to have the same preconditions as the query specifies. Furthermore, the Web service should provide the same set of effects that the query specifies. Checking such qualities require logical reasoning about preconditions and effects of Web services, which is a computationally expensive task.

In this thesis, we only deal with first type of the matching. However, we have some concrete future plans for behavioral matching of Web services too, for more details see Section 8.2. Also, note that behavioral matching usually tends to be a more expensive task and often it is wise to start with non-behavioral matching of Web services and once there are a reasonable number of matching candidates available, start doing the behavioral matching. Note that the matches (best matches) found and recognized through behavioral matching are necessarily a subset of the Web services recognized through non-behavioral matching process. If there are relatively small number of matches it is even possible to
6.1 SOFTWARE MATCHING

Software matching is the process of comparing two software artifacts or a software artifact and a query and determining their similarity. Similarity is not a rigid attribute, often different levels of similarity can be defined. Software matching mechanisms have diverse applications. One immediate usage would be searching and retrieving an existing piece of software with specified sets of characteristics that satisfies a certain user’s requirements.

To perform such comparisons, following formalisms and techniques are required:

- **Formalism to Model Software Artifacts:** An appropriate model is required to specify the attributes of software artifacts. Such a model should be in such a level of abstraction that provides enough information suitable for searching and finding those software artifacts efficiently.

  For Web services, as shown in Subsection 5.2.2, we find PIA an intuitive model for specifying their characteristics.

- **Formalism for Specifying a Query:** Obviously user of a matching system is not necessarily able to specify all the details of a desired piece of software. Thus, it is required to specify queries in a somewhat higher level of abstraction than the software artifact itself. Furthermore, extremely detailed specification of the required software
artifact may omit some results from the set of matches that could have possibly been considered acceptable matches.

In Web services matching, we find EPD and MPD, the interface models introduced in Section 2.3 and 2.2, quite helpful; they are in an appropriate level of abstraction for specifying queries in this context. EPD and MPD are both stateless interfaces and, as such, incapable of specifying different execution states of a system, which is not desired in this context either. In Subsection 6.1.1, we show how we can use EPD/MPD in this respect.

- **Matching Relations:** Depending on whether the comparison is happening between two sets of formalisms, for the case where a query is compared against a software artifact, or is happening in the context of a particular formalism, for the case where two software artifacts are being compared, we need to have formalisms to match either software artifacts against each other or a software artifact against a query. We call these kinds of relations *matching relations*.

In the context of Web service matching systems, we are dealing with matching relations that relate a query to a Web service’s description. In particular, we require a matching relation, capable of reasoning about similarities between EPDs/MPDs and PIAs. In Section 6.3, we show how such a relation can be defined.

### 6.1.1 Capturing User Requests

EPDs and MPDs are capable of modelling input and output ports of a system along with the dependencies among those ports. As such, they are abstract models suitable for high level specification of a system. In EPDs and MPDs, it is sufficient to specify the input and output messages of a system with possibly some dependencies among those messages. In this chapter, we use the terms “port” and “message” interchangeably. This is justified since EPDs and MPDs only care about the name and type of the ports, i.e. input or output. Messages in Web services provide these information.

As an example, consider EPD $F = \langle I_F, O_F, O_F^+, H_F, V_F \rangle$, which is a high level specifi-
6.1. SOFTWARE MATCHING

Consideration of a system:

\[
\begin{align*}
I_F &= \{\text{prod}_id, \text{prod}_name, \text{credit}_no\} \\
O_F &= \{\text{ref}_no, \text{err}_no\} \\
O'_F &= \{\text{ref}_no, \text{err}_no\} \\
H_F &= \emptyset \\
V_F &= \{v^1_F, v^2_F, v^3_F, v^4_F\} \\
v^1_F &= \langle (\text{prod}_id, \text{prod}_id), (\text{credit}_no, \text{err}_no) \rangle \\
v^2_F &= \langle (\text{prod}_id, \text{prod}_id), (\text{credit}_no, \text{ref}_no) \rangle \\
v^3_F &= \langle (\text{prod}_name, \text{prod}_name), (\text{credit}_no, \text{err}_no) \rangle \\
v^4_F &= \langle (\text{prod}_name, \text{prod}_name), (\text{credit}_no, \text{ref}_no) \rangle
\end{align*}
\]

In Figure 6.1, we have illustrated a pictorial representation of the above EPD. The dashed and dotted lines are meant to represent different dependency rules of \( F \). This figure in fact introduces the basis for modelling requests (queries) in a Web service matching system. Mapping such a pictorial representation of a query to an EPD interface is quite straightforward. In fact, the GUI of a user request in a Web services matching system can be built based on the idea of EPD’s blocks, ports, dependencies and connections.

MPD can also be used to model such requests. Recall that MPDs are only capable of storing sets of dependencies while EPDs can specify sequences of dependencies. In this way, the above EPD can be mapped as an MPD request by simply changing the sequences of dependencies to sets of dependencies. These two models provide the flexibility of specifying user request in different level of expressiveness. While EPDs can specify the temporal order of message exchanges, MPDs only specify the dependency between messages. Depending on user’s requirements, it is possible to use either MPD or EPD for capturing user requests.

Of course, it will be more meaningful if there exists an agreed-upon interpretation for the meaning of each port and also dependencies between the ports. Later in Section 6.4, we will make such interpretations more concrete in the context of Web service matching.

Also, note how the model in Figure 6.1 remotely resembles the PIA shown in Figure 3.12, on page 82. Notice the similarity between the sets of inputs and outputs in the two figures. In fact, we would like the EPD model in Figure 6.1 to be a valid query (request) for the automaton in Figure 3.12.
Figure 6.1: A pictorial representation of the EPD interface of Section 6.1.1. This figure can also pictorially represent a MPD with the same inputs, outputs and dependency sets. The numbers on top of the dependency pairs represent different dependency rules.

In the coming sections we will introduce *satisfiability relations*, which enable us to reason about such resemblance, e.g., between EPDs and PIAs.

## 6.2 Satisfiability Relations

Among interfaces belonging to a certain interface type, e.g. EPD or PIA, refinement is defined and provides a similarity measurement criteria. Dealing with interfaces belonging to different types of interfaces, it is desirable to have a criteria for measuring their *similarities*. Of course, we notice that different interface types tend to have different expressiveness capabilities and thus precise comparison of two different models may turn out to be irrelevant. However, often it is possible to define and provide an intuitive correlation between two different models such that it provides some criteria for similarity measurements. *Satisfiability* is such a correlation.

**Definition 6.1** Satisfiability relation $\Omega$ between two types of interfaces $A$ and $B$, which are either ordinary or weak interface types, is represented by $\triangleright^\Omega_{(A\rightarrow B)}$. Then, $a_1$, an instance of $A$, omega satisfies $b_1$, an instance of $B$, represented as $a_1 \triangleright^\Omega_{(A\rightarrow B)} b_1$. ■
6.3. SATISFIABILITY BETWEEN PIAS AND EPDS

$a \xrightarrow{\Omega} (A \rightarrow B) b$ is read as: “interface $a$ omega satisfies interface $b$”.

In the next section, we provide two satisfiability relations defined between PIAs and EPDs, i.e. weak and strong satisfiability.

6.3 Satisfiability between PIAs and EPDs

In this section, we provide two satisfiability relations between PIA and EPD. We first define weak satisfiability, $\llbracket (\text{PIA} \rightarrow \text{EPD}) \rrbracket^\text{Weak}$, and then define $\llbracket (\text{PIA} \rightarrow \text{EPD}) \rrbracket^\text{Strong}$ satisfiability based on weak satisfiability.

**Definition 6.2** A PIA, $P = (V_P^{\text{abst}}, V_P^{\text{init}}, V_P^{\text{comp}}, A_P^{\text{I simp}}, A_P^{\text{O simp}}, A_P^{\text{H simp}}, A_P^{\text{comp}}, \tau_P^{\text{abst}}, \phi_P)$, weak satisfies an EPD interface $F = (I_F, O_F, O_F^+, H_F, V_F)$, $P \llbracket (\text{PIA} \rightarrow \text{EPD}) \rrbracket^\text{Weak} F$, if all of the following conditions are true:

- $I_F \subseteq A_P^{\text{I simp}}$
- $O_F \subseteq A_P^{\text{O simp}}$
- For each dependency rule $\nu = \{(a_1, b_1), (a_2, b_2), \ldots, (a_n, b_n)\} \in V_F$, there exists an execution $E \in \text{Exec}(P)$, (called satisfaction fragment), with $\text{sched}(E) = (x_1, x_2, \ldots, x_m)$, such that all of the following conditions are true:
  - For all $(a_i, b_i) \in \nu$:
    * $\text{pos}_{\text{sched}(E)}(a_i) \neq \emptyset$
    * $\text{pos}_{\text{sched}(E)}(b_i) \neq \emptyset$
    * $\text{pos}_{\text{sched}(E)}(a_i) \leq \text{pos}_{\text{sched}(E)}(b_i)$
  - For all $(a_j, b_j)$ in $\nu$ and all $i$ s such that $i < j$:
    * $\text{pos}_{\text{sched}(E)}(b_i) < \text{pos}_{\text{sched}(E)}(b_j)$
    * $\text{pos}_{\text{sched}(E)}(a_i) < \text{pos}_{\text{sched}(E)}(a_j)$

$^1$An execution of a PIA is an execution fragment starting from an initial state and terminating in a terminating state, see Definition 3.5.
This definition states that if $P \text{Weak}$ satisfies $F$, $P \uparrow_{(PIA-EPD)}^{Weak} F$, then for each dependency rule in $\mathcal{V}_F$ there exists an execution in $P$ that simulates those dependencies, i.e., the first element of a certain dependency definitely happens before the second element. As an example, for a certain dependency $(a, b)$ where $a \in \mathcal{A}^{I, simp}_P$ and $b \in \mathcal{A}^{O, simp}_P$, the corresponding satisfaction fragment, would have $a$ before $b$ in its schedule, which can be interpreted as function call (or service invocation) $a$ happens before the result of such function call, $b$, is returned back to the environment. Furthermore, the sequence of dependency pairs in a certain dependency rule is respected in such an execution.

As for identity dependencies, their presence in a dependency rule implies the existence of that certain port in $P$. Identity dependencies are useful when considering that some Web services can receive multiple inputs (in a specific order) and return back one output. Since one output can not be influenced by more than one input we group the last input element and the output in one dependency pair; for the rest of inputs, use identity dependencies to model them. Since EPDs can specify the order of dependencies it is possible to properly specify the order of inputs through dependency pairs.

We can make the weak satisfiability relation more precise by restricting the satisfaction fragments for each dependency rule to contain only the input and output ports of the corresponding dependency rule. We call this satisfiability relation $Strong$ satisfiability.

**Definition 6.3** A PIA, $P = (V^\text{abst}_P, V^\text{init}_P, V^\text{comp}_P, \mathcal{A}^I_P, \mathcal{A}^O_P, \mathcal{A}^H_P, \tau^\text{abst}_P, \phi_P)$, $Strong$ satisfies an EPD interface $F = (I_F, O_F^+, O_F^-, H_F, \mathcal{V}_F), P \uparrow_{(PIA-EPD)}^{Strong} F$, if $P \uparrow_{(PIA-EPD)}^{Weak} F$ and furthermore, for each dependency rule $\nu = \{(a_1, b_1), (a_2, b_2), \ldots, (a_n, b_n)\}$, where $\nu \in \mathcal{V}_F$, and its corresponding satisfaction fragment, $E \in \text{Exec}(P)$, where $\text{sched}(E) = \langle x_1, x_2, \ldots, x_m \rangle$, one of the following is true:

- $x_k = a_i \quad (1 \leq i \leq n) \land (1 \leq k \leq m)$
- $x_k = b_i \quad (1 \leq i \leq n) \land (1 \leq k \leq m)$
- $x_k \in \mathcal{A}^H_P$

$^2$Dependency $(a, b)$ is an identity dependency if $a = b$, see Section 2.3 for more information.
As an example, consider the PIA in Figure 6.2. This PIA \textit{Strong} satisfies the EPD \(F\), in Figure 6.1. Notice that \texttt{CompPay} contains all \(F\)'s input and output ports and also for each dependency rule there exists an execution which satisfies that dependency. As an example, the execution \(\langle 1, \text{prod\_name}\?, 2, \text{credit\_no}\?, 3, \text{cnd\_price}\; ;, 4, \text{err\_no}\!\!, 5 \rangle\), in Figure 6.2, is the execution that satisfies \(v_1^F\).

Considering two EPDs \(F\) and \(G\) and two PIAs \(P\) and \(Q\) satisfying \(F\) and \(G\) respectively. It is desirable that \(P \parallel Q\) to satisfy \(F \parallel G\). However, it is not guaranteed that such a satisfiability relation will hold for their composition. It can be observed that in absence of shared ports between \(F\) and \(G\), it is highly likely that \(P \parallel Q\) can satisfy \(F \parallel G\).

It is desirable, whenever possible, to define satisfiability relations in a compositional manner, as described above.
6.3.1 Strong Satisfiability Algorithm

In this subsection we provide an efficient algorithm that checks whether a given EPD can be *Strong* satisfied by a PIA. Our proposed algorithm is based on dynamic programming\(^3\) paradigm where the solution for a problem is computed by solving some subproblems in a bottom-up fashion.

In dynamic programming, we always look for the structure of an optimal solution which itself leads to solving some subproblems. In case of our problem, consider the dependency rule \(\nu = \langle(a_1, b_1), (a_2, b_2), \ldots, (a_n, b_n)\rangle\), as specified in Definition 6.2, and an arbitrary execution \(E = \langle v_1, x_1, v_2, x_2, \ldots, v_{m+1} \rangle\). If \(E\) is a satisfaction fragment for \(\nu\) then there should exist a \(v_l\), where \(0 \leq l < (m + 1)\), such that \(E' = \langle v_1, x_1, v_2, x_2, \ldots, v_l \rangle\) satisfies \(\nu' = \langle(a_1, b_1), (a_2, b_2), \ldots, (a_{n-1}, b_{n-1})\rangle\). This relation suggests that we can decompose the problem to smaller subproblems.

To make the structure of such subproblems more concrete, consider notation \(f(v, i, s)\), where \(v\) denotes a state in \(P\), and \(1 \leq i \leq n\), represents one of the dependency pairs of \(\nu\) and \(s\) is a set which is a member of \(S = \{s_0, s_1, s_2, \ldots, s_k\}\). \(k\) is the number of distinct members in \(\text{Domain}(\nu)\) and we have \(s_k = \{a_1, a_2, \ldots, a_k\}\) and all such \(a\)'s are exactly the first \(k\) distinct \(a\)'s that appear in \(\nu\). Furthermore, \(s_0 = \emptyset\). The value of \(f(v, i, s)\) is either 0 or 1. Zero value would indicate that at state \(v\) the dependency pair \((a_i, b_i)\) is not yet satisfied. A value of 1, on the other hand, suggests that at state \(v\) dependency pair \((a_i, b_i)\) is satisfied and \(a_i \in s\).

For dependency rule \(\nu \in \mathcal{V}_F\), the number of \(f(v, i, s)\)'s to be computed are \(|V_P| \times n \times (|\text{set}(\text{Domain}(\nu))|)\). In fact at each state all possibilities for that state are considered. \(P\) has a satisfaction fragment for \(\nu\), if there exists a \(u \in V_P^{\text{term}}\) such that \(f(u, n, s_k) = 1\). In other words, at state \(u\) we have seen all actions belonging to \(\text{Domain}(\nu)\) and have also received all actions in \(\text{Range}(\nu)\) in an appropriate order that satisfies the order of \(\nu\)'s

---

\(^3\)Dynamic programming paradigm is helpful in solving problems where the main problem can be decomposed into some subproblems. In dynamic programming the result of solving such subproblems happen to be needed in different stages; using conventional divide and conquer paradigm, we have to solve a single problem multiple times while in dynamic programming we store such results and use them whenever necessary. Usually dynamic programming subproblems can be stated in a recursive fashion, their computation, however, happens in a bottom-up fashion based on the recursive definition of such subproblems. For more information on dynamic programming paradigm refer to [THCS01].
elements.

We can populate all $f(v, i, s)$ s in a bottom-up manner. Starting from the initial state we can compute the $f$ values for other states reachable from initial state based on the values of initial state, which is 0.

To do such a bottom-up computation, we need to have all necessary $f$ values pre-computed. To achieve that, we actually need to know the topological order of states. In other words, it is required to have a sequence of states such that steps are issued only from the states with lower order in the sequence to the states with higher order in the sequence. An acyclic directed graph can be topologically sorted in time linear in the size of the sum of its edges and vertices.\footnote{\textcolor{red}{The topological sort, as described in [THCS01], uses depth first search on a directed graph to sort vertices in a topological order.}} To use topological sort, we should map a PIA into a directed graph representation. Once the topological sort is carried out, it will also be the topological order of states in the corresponding PIA. Obviously, $V_P^{\text{init}}$ is the first member of such a sort.

$P$ can be mapped into a directed acyclic graph by mapping states as vertices of a graph and steps as simple edges without labels. In other words, if there are multiple steps between two states, they will be mapped as a single edge of the graph. Furthermore, we require the resulting graph to be acyclic. Any PIA having loop states can be trivially modified not to have such states. For loop states belonging to $V_P^{\text{loop}}$, the steps initiating from such states, which go back to the initial state, are simply redirected to a new terminating state. For loop states not belonging to $V_P^{\text{loop}}$, all steps initiating from such states that create cycles are removed.

To compute $f$ values, we first set all possible $f$ values to zero: $f(l, i, s) = 0$, where $l \in V_P$, $1 \leq i \leq n$ and $s \in S$. Having the states of $P$ sorted, we can compute the $f$ values in a bottom-up fashion. At any state $v$, where $v$ and all other states in lower order of $v$ have had their $f$ values computed, we consider each step $(v, t, u)$, such that $t \in A_P^{\text{simp}}$, against all $f(v, i, s)$ values and then determine the corresponding values for $f$ values for $u$. 
Given step \((v, t, u)\) and \(f(v, i, s)\), following are possible \(f\) values for \(u\):

\[
\begin{align*}
\{f(u, i, s + \{t\}) = 1 & \quad \text{if } (t \notin s) \land (t \in \text{Domain}(\nu)) \land (f(v, i, s) = 1) \\
\{f(u, i + 1, s) = 1 & \quad \text{if } \text{Pos}_{\text{Range}(\nu)}(t) = i + 1 \land (a(\text{Pos}_{\text{Range}(\nu)}(t)) \in s) \\
\{f(u, i, s) = f(v, i, s) & \quad \text{if } (t \in A_{\text{H.simp}}) \land (f(v, i, s) = 1)
\end{align*}
\]

The first possible value for \(f\) is created when action \(t\) belongs to \(\nu\)'s domain and is not present in set \(s\). Furthermore, all earlier domain elements of \(\nu\), that should appear before \(t\), have been received and it is the right time to receive \(t\). The second possibility deals with the situation when \(t\) is an action belonging to \(\nu\)'s range and its corresponding element in \(\nu\)'s domain is already received, i.e. it is a member of \(s\). Furthermore, all dependencies happening before that dependency in \(\nu\) are satisfied and thus we can announce the dependency pair on \(t\) satisfied. The third possibility happens when we receive an internal action and we can transfer the value of \(f\) to the state that \(t\) ends in.

Any other combination of values for \(t\) does not affect \(f\) values of \(u\). As an example, if there is a \(t\) such that it is neither in domain nor in range of \(\nu\) and also it is not a internal action, then such a \(t\) would not change \(f\) values. In fact, it is an invalid step. As mentioned earlier, if there is an execution \(E\) such that it satisfies \(\nu\) then for the terminating state of \(E\), suppose it is \(e\), we will have \(f(e, n, \text{set(Domain}(s))) = 1\).

The algorithm in Figure 6.3 computes all \(f\) values in a bottom-up fashion starting from the initial state. However, before using this algorithm, as mentioned earlier, it is required to topologically sort the set of states \(V_P\).

\[
V[\] = \text{TOPOLOGICAL.SORT}(P)
\]

Function \(\text{TOPOLOGICAL.SORT}()\) first transforms \(P\) into a directed graph and then sorts its states. This extra transformation does not cause extra complexity to topological sort execution. \(V = \langle v_1, v_2, \ldots, v_p \rangle\) is an array containing all states in \(V_P\) sorted in a topological order. \(V[\] is passed to algorithm \(\text{SAT\_PIA\_EPD\_STRONG}\) along with \(P\) itself and \(\nu\). We let the size of \(\text{set(Domain}(\nu))\) be \(k\) and assume that all \(s_i \in S\), where \(0 \leq i \leq k\), are already computed and can be used by the algorithm.
6.3. SATISFIABILITY BETWEEN PIAS AND EPDS

*SAT\_PIA\_EPD\_STRONG* starts its job by initializing all $f(v_i, 0, s_0)$ values to 1; $f(v_i, 0, s_0)$ values in fact are helper values that enable us to compute $f$ properly. This is especially useful when we are dealing with the situation in which the algorithm has not come across any members of $Range(\nu)$ and we need to maintain the information about already visited actions that are member of $Domain(\nu)$. The rest of the algorithm is quite straightforward and does the computation according to the definition we have already provided for $f$.

Using $f$ values, we can refer to the states belonging to $V_P^\text{term}$ and check whether the automaton can *Strong* satisfy $\nu$. Any terminating state $u$ with $f(u, n, \text{set}(\text{Domain}(s))) = 1$ suggests that there exists a satisfaction fragment for $\nu$. Notice that by backtracking on states and starting from such $u$ we can actually compute the corresponding satisfaction fragment. We do not provide that algorithm in this thesis though.

Complexity of this algorithm is obviously $|V_P| \times |\nu|^2 \times |\tau_P|$. Notice that we can slightly modify this algorithm to compute all $f$ values for all dependency rules in $V_F$ in one run. This can be accomplished by keeping track of different versions of $f$ for different dependency rules. However, in presence of large number of dependency rules, a large space complexity is required. The trade-off then would be between time and space efficiency.

Notice that if a request can be decomposed into multiple composable EPDs it is possible to inspect the satisfiability of each EPD independently and thus in a concurrent manner. This is since PIAs are both commutative and associative and as such a set of composable PIAs can be composed in arbitrary order. As soon as two PIAs are recognized to satisfy their corresponding EPDs they can be composed and stored until the next satisfying PIAs are identified.

The algorithm for $\triangleright^{\text{Weak}}_{\text{PIA} \rightarrow \text{EPD}}$ can be obtained by slightly modifying this algorithm. In weak satisfiability we should be able to let the PIA accept actions that are not present in the corresponding EPD. To achieve that, it suffices to deal with those extra actions in the same way that we deal with internal actions in the algorithm.
Algorithm SAT_PIA_EPD_STRONG ($P$, $V = (v_1, v_2, \ldots, v_p)$, $\nu = ((a_1, b_1), (a_2, b_2), \ldots, (a_n, b_n))$):

Variables: $i = 0, j = 0$;

\[ l = 0; \]
\[ f(p, n + 1, |set(Domain(\nu))| + 1) = 0; \]

begin

for $i = 1 \ldots p$ do

\[ f(v_i, 0, s_0) = 1; \]

endfor

for $i = 1 \ldots p$ do

for $j = 0 \ldots n$ do

for $l = 0 \ldots |set(Domain(\nu))|$ do

foreach $(v_i, t, u) \in \tau_P$ do

if $(t \notin s_l)$ and $(t \in Domain(\nu))$ and $(f(v_i, j, s_l) == 1)$ and $\left( a(\text{Pos}(\text{Range}(\nu))(t)-1) \in s \right)$ and $\left( a(\text{Pos}(\text{Range}(\nu))(t)+1) \notin s \right)$ then

\[ f(u, j, s_l + \{t\}) = 1; \]

endif

if $(\text{Pos}_{\text{Range}(\nu)}(t) == j + 1)$ and $(a(\text{Pos}_{\text{Range}(\nu)}(t)) \in s)$ and $(t \in \text{Range}(\nu))$ and $(f(v_i, j, s_l) == 1)$ then

\[ f(u, j + 1, s_l) = 1; \]

endif

if $(t \in A_P^{H,Simp})$ and $(f(v_i, j, s_l) == 1)$ then

\[ f(u, j, s_l) = 1; \]

endif

endforeach

endfor

endfor

return $f()$;

end

Figure 6.3: The outline of the algorithm for checking Strong satisfiability between PIA and EPD. The return value would be the table $f()$ which contains necessary information for determining satisfiability and the satisfaction fragment.
6.4 Web Services Satisfaction

In a Web services discovery system we need to receive users’ requests in an appropriate level of abstraction. We can not expect the users to specify exact details of their desired Web services. There are two major reasons for this, first, they can not be always expected to know the exact flow of execution in a composite web service; especially in case of complicated services. Second, an exact specification may ease the task of searching but at the same time may eliminate some valid web services from the search result. As mentioned earlier, stateless interfaces such as EPDs and MPDs tend to be appropriate abstract models for specification of requests in such discovery systems.

On the other hand, in Chapter 5, we showed how PIA is an appropriate formalism for modelling Web services. Thus, we can use satisfiability relations such as $\text{strong}_{(\text{PIA} \rightarrow \text{EPD})}$ to inspect the matching relation between a request and a Web service. A certain request can be specified as composition of multiple EPDs and then all such EPDs can be searched for their satisfying PIAs in a parallel manner. In the context of Web services, this means the capability of issuing concurrent searches over the Web or different repositories of Web services information. Ultimately the composition of search results can be checked for satisfiability.

We choose to use EPD as the formalism for modelling user requests and follow $\text{strong}_{(\text{PIA} \rightarrow \text{EPD})}$ relation for checking the matching between requests and Web services.

Although $\text{strong}_{(\text{PIA} \rightarrow \text{EPD})}$ satisfiability relation is quite helpful for recognizing the matching Web services; but we are also interested in enhancing that relation such that it can check for some inherent characteristics of Web services too. Recall that we chose OWL-S as the specification language for composite Web services; one of OWL-S’s components is the grounding class which maps simple Web services into their corresponding WSDL operations and messages. WSDL messages belonging to a certain operation follow a specific message exchange patterns, as specified in Table 4.1, on page 98. We would like these patterns to influence the process of Web services matching. In specific, we would like to check whether a certain PIA (which represents a Web service or composition of a set of Web services) complies with the specification that the user has specified in an EPD. To do such a checking we should define a correlation between EPDs’ dependencies and different message exchange patterns.
Table 6.1: Correlation of dependency pairs belonging to a dependency rule and different WSDL “message exchange patterns”. \((a_i, b_i)\) is an arbitrary dependency pair belonging to one of dependency rules of EPD \(F\).

<table>
<thead>
<tr>
<th>Dependency Pair</th>
<th>WSDL Message Exchange Patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a_i = b_i)</td>
<td>All Patterns</td>
</tr>
<tr>
<td>(a_i \in I_F \land b_i \in O_F)</td>
<td>“In-Out” and “In-Multi-Out”</td>
</tr>
<tr>
<td>(a_i \in O_F \land b_i \in I_F)</td>
<td>“Out-In”, “Asynchronous Out-In” and “Out-Multi-In”</td>
</tr>
</tbody>
</table>

Consider \(\nu = \langle (a_1, b_1), (a_2, b_2), \ldots, (a_n, b_n) \rangle\) a dependency rule of EPD \(F\), as defined in Definition 6.3. For each \((a_i, b_i) \in \nu\), it is either the case that \(a_i = b_i\), which implies \((a_i, b_i)\) is an identity dependency pair, or \(a_i \neq b_i\), which implies that one of \(a_i\) or \(b_i\) is an input element and the other one is an output element. As such, a dependency pair can be correlated with WSDL message exchange patterns. Table 6.1 illustrates such correlations.

Table 6.1 illustrates possible “message exchange patterns” for different types of dependency pairs. Depending on the value and the type of the source and destination of a dependency pair, they can be satisfied by certain message exchange patterns. Notice that if \(a_i = b_i\) then all patterns are valid. This is because, as mentioned earlier, we can use identity dependencies to model Web services that have multiple inputs and a single output. Conversely, it is possible to have a Web service operation that has one input and multiple outputs.

Considering the cases where \(a_i \neq b_i\), as an example the case where “\(a_i \in I_F \land b_i \in O_F\)”, then for dependency pair \((a_i, b_i)\), there should exist actions in an execution fragment of the satisfying PIA that satisfies that particular pair. Furthermore, it should be the case that such actions are mappings for a Web service operation that have either “In-Out” or “In-Multi-Out” grounding classes. Furthermore, both \(a_i\) and \(b_i\) should be mapped to the same WSDL operation through that grounding class. As an example, assume that \(a_i\) is associated to the input message of a WSDL operation \(x\), where \(x\)’s message exchange pattern is “In-Out”, and \(b_i\) is mapped to operation \(y\), which also has “In-Out” message exchange pattern; such an execution then would not be able to satisfy the dependency pair \((a_i, b_i)\) because \(x \neq y\). In other words, a dependency pair consisting of distinct elements should simulate
a WSDL operation and thus it is expected that both elements of a certain dependency pair belong to the same operation, i.e. two elements simulate input and output messages of a single operation.

Considering the algorithm in Figure 6.3, we can enhance it in such a way that we can check the above extra satisfaction conditions for Web services satisfiability. The modification for such an enhancement can be done in a straightforward manner by retrieving associated grounding information of each action, in the satisfying PIA, and checking whether it has a valid operation type with proper message exchange pattern. Also, it should be checked whether a certain port, participating in a dependency pair, is associated with the same WSDL operation that the other port in that pair is associated with.

6.5 Semantic Matching of Web Services

In the previous section we proposed a method for discovery of Web services. Our proposed method takes advantage of \( \uparrow^{\text{Strong}}_{(\text{PIA} \rightarrow \text{EPD})} \) satisfiability relation. Note that in the previous section we had this important assumption that the vocabularies used for the sets of input and output elements in EPDs and PIAs are the same. However, this assumption is not always a realistic assumption; the reason mainly being that the requester of a certain Web service does not necessarily know the name of the input and output messages of the desired Web service. In fact, it is more probable that requesters have a vague idea of their desired Web service and the name and content of messages belonging to such a Web service.

In Section 5.1 we introduced semantic Web and some related semantic Web standards such as \texttt{OWL-S}. Recall that semantic Web standards are capable of comparing documents in a semantic manner. In the context of Web service discovery systems, we would like to take advantage of such semantic Web standards’ capabilities. It is desirable for Web services requesters to be able to specify the input and output elements of their request in a flexible and not necessarily precise manner. As an example, if a requester specifies a request with the input element “ISBN” and the output element “Title” then we would like a Web service with the input element “BookNumber” and output element “BookName” to satisfy such a request.

In fact, different levels of semantic similarities among two \texttt{OWL-S} enabled messages can
be defined. In this thesis, we adhere to the similarity levels defined in [PKPS02]. In [PKPS02], authors suggest an algorithm that receives a request and works on an ontology of Web services, specified in $\text{DAML-S}$\textsuperscript{5} [BHL+02], to rank the matching Web services. In the following, we provide a brief description of authors’ proposed semantic matching algorithm. In this thesis, we are only interested in their proposed levels of semantic matching between messages and show how it can be used to enhance our discovery mechanism.

The proposed semantic matching algorithm receives the requests in forms of a set of inputs and outputs. The vocabulary used for specifying such input and output elements should be valid vocabularies in the ontology on which the algorithm works on. The request can be considered as a simple processes representing the glass-box view of the desired Web service, see Section 5.2.1 for more information on glass-box view.\textsuperscript{6}

Once such a request is specified, the semantic matching algorithm works on the repository of existing Web services and by comparing the sets of input and output messages against advertised Web services’ messages, evaluate their semantic level of matching. For a certain request, any satisfying Web service should satisfy all of the request’s input and output messages.

The semantic matching algorithm uses different approaches for input and output matching. The output messages of the requests are matched against the output messages of the advertised Web services’ messages and conversely the input messages of the advertised Web services are matched against the request’s input messages.

Table 6.2 illustrates how the level of matching can be specified for input and output messages. Two semantic messages are equal if they refer to the same object or are declared explicitly in the ontology to be equal. “subclassOf” relation, on the other hand, represents the immediate subclass relationship between two classes as described in $\text{OWL}$, see Section 5.1.3 for more details. “subsumes” relation happens when two classes do not have an immediate subclass relationship; however, they belong to the same branch of concepts in an ontology, and furthermore, one of the classes can be assumed to be logically containable

\textsuperscript{5}$\text{DAML-S}$ is predecessor of $\text{OWL-S}$; it similarly models Web services as processes. The approach proposed in [PKPS02] for semantic matching of Web services remains consistent and applicable to $\text{OWL-S}$ ontologies too.

\textsuperscript{6} Notice that such requests are consistent with EPD formalism which we propose for capturing the user requests. In fact, an EPD without any dependency rules is similar to such requests.
6.5. SEMANTIC MATCHING OF WEB SERVICES

<table>
<thead>
<tr>
<th>Request/Advertisement</th>
<th>Input Messages</th>
<th>Output Messages</th>
</tr>
</thead>
<tbody>
<tr>
<td>R = A</td>
<td>“exact”</td>
<td>“exact”</td>
</tr>
<tr>
<td>R subclassOf A</td>
<td>“subsumes”</td>
<td>“exact”</td>
</tr>
<tr>
<td>A subclassOf R</td>
<td>“exact”</td>
<td>“subsumes”</td>
</tr>
<tr>
<td>A subsumes R</td>
<td>“subsumes”</td>
<td>“plugIn”</td>
</tr>
<tr>
<td>R subsumes A</td>
<td>“plugIn”</td>
<td>“subsumes”</td>
</tr>
<tr>
<td>Otherwise</td>
<td>“fail”</td>
<td>“fail”</td>
</tr>
</tbody>
</table>

Table 6.2: Different levels of semantic matching between input and output messages.

in another one.

Four levels of matching are defined in Table 6.2: exact, plug in, subsumes and fail. They suggest different levels of semantic similarity in a decreasing order. “Exact” match happens when the request and process messages belong to the same class of an ontology. Furthermore, a request’s output message that is subclass of a certain output message represents an exact matching. Conversely, an input message which is subclass of a request’s input message represents an exact match. Other levels of matching suggest weaker levels of similarities. Based on these levels of matching, a matching engine can be used to search over a repository of Web services and retrieve the best possible matches.

For a certain comparison between a request and an OWL-S process, each message in the request is checked against all of that process messages. Then, the message with maximum matching level in the process will be chosen. The matching message then is not considered in comparisons anymore. Once all of the request’s messages are matched in this way successfully the output level of matching is the minimum degree of matching among all output messages and similarly, input level of matching is the minimum degree of matching among all input messages.

In [PKPS02], the authors propose that level of matching for output messages to play a more important role than the input messages. They also reflect this idea in their algorithm. We adhere to this observation since output messages tend to represent functionality of processes more effectively. Any matching process, however, is required not to have any messages in the request with matching level equal to “fail”.
Satisfiability process can be parameterized to work with different levels of matching. In fact, we may no longer look for the exact appearance of a certain port of an EPD in a PIA. Instead, an acceptable level of semantic matching would suffice. Notice that the acceptable level of matching could be considered as a parameter to the satisfiability algorithm. As long as all ports in a request EPD has counterpart actions in the satisfying PIA, with an acceptable level of semantic matching, our satisfiability algorithm can proceed and check for existence of satisfaction fragment in that PIA.

In this way, our Web service discovery approach can start its job with PIAs that have a high level of semantic matching with the request, and inspect whether they satisfy that request. In case those PIAs do not satisfy the request, the algorithm would proceed with PIAs with lower level of semantic matching.

In next chapter, where we describe Web services composition algorithm, we further elaborate on how such a semantic matching mechanism fits in the context of our Web service discovery approach.
Chapter 7

Composing Web Services

While simple requests can be satisfied by simple Web services, it may be required to compose multiple Web services to satisfy more complicated requests. In this chapter we propose an algorithm that is capable of receiving complicated requests and by working on a repository of Web services and composing them satisfies such requests.

We also show how we can decompose a complicated request into some constituent sub-requests that can be more easily satisfied. We further show different operations, other than composition, which are required to specify the correlation of such sub-requests. Such sub-requests can be checked for satisfiability in a concurrent manner, as shown in the previous chapter, and then later the composition of satisfying search requests can be checked for satisfiability of the original search request.

Before describing our approach for composition of Web services, we briefly describe some other approaches for composition of Web services. Our approach, however, is in contrast with other approaches in different major areas. The most distinguishing differences being: first, it is based on a coherent set of formalisms, second, it is capable of determining the similarity of instances belonging to different models. And lastly, the presence of an explicit method for composition of Web services.

The remainder of this chapter is organized as following. In Section 7.1, we describe some of the related works in the area of Web services discovery and composition. Section 7.2 describes our composition algorithm along with the necessary preprocessings that should be performed on the requests before submitting them to the composition algorithm. At the
end, in Section 7.3, we show how for a composite Web service, which is discovered/created by the composition algorithm, a glue code, which is capable of executing that service, can be automatically generated.

7.1 Related Works

There has been different efforts to model composite Web services. The main focus has been on how composite Web services can be represented in a meaningful, expressive manner; and more importantly how such Web services can be created. In this section we are basically interested in studying some of the more comprehensive approaches, also approaches that are relevant to our proposed approach. By analyzing these approaches we depict how our approach is different from other approaches and what is the real contribution of our approach. However, there are some other interesting efforts that attack the problem from a different perspective. One of the interesting approaches in this respect is introduced in [MS02] where the problem of synthesizing composite Web services is reduced to a planning problem. Authors take advantage of the semantics of situation calculus [Rei01] along with Golog language [LRL+97] to create abstract programs capable of dynamically composing Web services. This approach works on a set of DAML-S specified Web services to accomplish its task.

In general, different approaches discussed in the following subsections often do not differentiate between two distinguished tasks of discovery and composition. We believe these two concerns should be separated since they are basically two separate problems. Also, it has become clear to us that some of the approaches only manage to show how a formalism can be used to model composite Web services and do not bother about showing how they can be discovered or constructed.

Another main issue is to distinguish between the run time and design time specification of composite Web services. While some approaches clearly distinguish the difference, others do not. This is important since these two methods imply two sets of different problems to be solved.

Also, if we define Web services composition as specification of control flows and data flows between multiple Web services, then many approaches do not provide accurate spec-
7.1. RELATED WORKS

ification of dataflows inside composite Web services. This is as opposed to control flows that are usually better studied and analyzed.

7.1.1 Petri Net Approach

This approach [NM02], relies on semantics of Petri Nets for modelling and composition of Web services. The authors' goal is to create a set of formalisms capable of composition and verification of composite Web services. They start with DAML-S [BHL+02] as the semantic specification of Web services. Situation calculus is used as a logical model which captures the behaviors of DAML-S’s processes and control constructs. However, situation calculus does not provide any semantics for how the composition itself should happen.

By mapping situation calculus representation of Web services into their Petri Nets equivalents, authors show how Petri Nets semantics can be used to analyze the behavior of Web services. Petri Nets are shown to be capable of simulating various workflow patterns and operations and properties of composite Web services, such as composition and validation, are accomplished by using properties of their counter part Petri Nets structures. The notion of reachability in Petri Nets is used to reason about properties of Web services.

This approach fails to provide a composition mechanism and only sequential execution of Web services can be defined. What we consider real composite processes, having various control constructs being involved, and authors call it all possible interleavings, can not be created in this approach.

In general the proposed formalism seems to be more successful in simulating the behavior of already existing composite Web services. Another drawback is the fact that the resulting Petri Nets mappings of Web services are in a rather high level of abstraction. Input and output messages of Web services, for example, are not involved in mappings and consequently it is not possible to reason about them. The authors are mainly concerned with mapping DAML-S processes and their interconnecting control constructs.

However, Petri Nets seems to be a useful tool for execution of composite Web services. The notion of place and token naturally fit the workflow-based execution of Web services.
CHAPTER 7. COMPOSING WEB SERVICES

7.1.2 Finite State Machine Approach

In this approach [BCG+03], different sets of formalisms are employed to create the theoretical grounds for automatic composition of Web services at design time. Finite state machines, FSMs, are used as the formalism for conceptually modelling the behavior of Web services. Based on the semantics of FSM, authors propose an XML-based language, WSTL[BRSM03], Web Services Transition Language, which is capable of expressing specifications of composite Web services. The specification is basically a mapping from an equivalent FSM representation of the Web services. The transitions happen by means of WSDL messages and as such WSTL is built on top of WSDL.

TFSM, Timed FSMs, are used to specify the timing constraints of the message passing. The authors suggest using the timing constraints on input and output messages. Time constraint for an input message represents the maximum amount of time that a Web service would wait in a specific state for an input message. On the other hand, the output timing constraint would imply the maximum amount of time that the Web service would require to carry out an operation and create an output message. While TFSM seems a reasonable model, it is not clear how and from where we can provide the required timing constraint information. Basically non-functional requirements of Web services, including timing constraints, are difficult to be measured and specified. Moreover, there is no core Web service standard specifying these kinds of attributes. It seems to us it is rather immature to try to model a sophisticated attribute, i.e., timing constraints, via a single number. Response times usually depend on various parameters, including time of the day the service is requested, the platform it is being run, bandwidth of the service requester’s network and etc., which can not be specified through a single number.

In this approach, FSM is used in two scenarios to represent two different concepts, namely, external schema and internal schema of composite Web services. External schemas are meant to capture the specification of requests for the composition system, while the internal schemas model the real execution scenario for a set of collaborative Web services in a composite Web service. Mealy FSM \(^1\) is used to model internal schemas. Mealy FSMs produce outputs on each transition of a certain FSM; these outputs can be used to specify the identity (also the instance identifier) of the Web service initiating that transition.

\(^1\)A finite state machine that produces an output for each transition.
Moreover, both external and internal schemas can be mapped to trees of execution that are more intuitive for modelling and illustrating the sequence of execution in a composite Web services.

*FSMs* do not provide any composition semantics, necessary for combining two Web services. The authors use *DPDL, Deterministic Propositional Dynamic Logic* [KT90], as a tool for composing Web services. *FSMs* are mappable to *DPDL* formulas. *DPDL* structure is mainly developed to reason about the structure and correctness of programs and happen to be helpful in the context of Web services composition. However, there is no enhanced composition techniques in *DPDL*. The composition simply happens by combining different predicates belonging to different *FSMs*, i.e., different Web services. In fact, there is no specific algorithm to choose and compose the existing Web services in a repository and integrate them towards a match, i.e., an external schema. The composition is exponential to the number of available Web services which seem to be unreasonable in presence of large number of Web services. For more details on composition approach, see [BCDG+03, BCG+03].

Also, the authors show how they can employ situation calculus to reason about properties of Web services from logical point of view. In [BCDGM03], authors show how situation axioms can be mapped to *DPDL* structures and thus the proposed satisfiability relations are applicable to this problem. However, the authors do not provide how they can define and populate situation calculus structures by existing set of formalisms. In fact, there is no structure presenting the preconditions and post conditions of the Web services. Any kind of reasoning about preconditions and effects of Web services can be only meaningful in a semantic enabled context, which authors do not deal with it. Moreover, the mapping from situation calculus to *DPDL* is rather ad hoc and incomplete; thus the proposed satisfiability relation seems to be incompetent.

There are some other drawbacks to this approach too. Firstly, *FSM* is not an enough expressive tool for modelling Web services. In specific, workflow patterns along with input and output behaviors of Web services are not captured by *FSM*. We believe that workflow patterns are fundamental to any proposed orchestration technique. The lack of expressivity in *FSM* leads to the necessity for various formalisms getting involved. High number of translations required, back and forth between different formalisms, make this approach
less intuitive and efficient. Also, often there are modelling redundancies; as an example, it seems that authors neglect the fact that WSDL represents the temporal order of messages in an operation. They unnecessarily model the temporal order of messages by timed FSM. Moreover the most recent version of WSDL introduces different operations, among them operations with multiple input or output messages. The proposed model breaks with this new WSDL specification. It seems to us that the main reason for this, is due to the fact that there does not exist any first-hand definition for input and output messages, which is key to modelling Web services’ behaviors.

Similar to other Web services composition paradigms, authors do not present any formal tool for capturing requests in a Web service composition system. In fact, external schemas, in forms of trees or FSMs, are supposed to model user’s requests; however, as we discussed in Chapter 6, user requests often are not clear and precise and thus any model capturing those requests should be able to deal with certain amount of sloppiness. This approach, on the other hand, relies on precise specification of the requests, i.e., precise external schemas. This implies that users have a priori knowledge of the existing Web services which is usually not the case; even if that is the case, then the necessity of a composition system would be undermined.

Another key problem in this model is the absence of parallelism in different parts of the formalism. The proposed formalism does not propose any formalism for expressing concurrent execution of Web services, while all of orchestration formats, e.g. OWL-S, BPEL4WS, have specific workflow patterns for specification of parallelism. Trees and FSMs on the other hand, are not inherently suitable for specifying parallelism.

### 7.1.3 State Charts Approach

In this approach the authors use statecharts [Har87] to propose and implement a method for specification and execution of composite Web Services. State charts have been extensively used to specify the structure and temporal behaviors of systems. ² Explicit syntactical notations exist for expressing concepts such as concurrency and complex states; as such they tend to be a suitable model for specifying complex systems. Based on state charts

---

²UML adapts state charts to model the behaviors of UML classes and use cases.
7.1. RELATED WORKS

The authors introduce an architecture, SELF-SERV [BDSN02, BDFP01], for dynamic discovery and execution of Web services. This approach relies on the concept of service communities [BDSN02]. A service community describes the capabilities of a desired service without referring to any specific Web service instance. Each community can have multiple subscribers registered to it and it is possible to choose each of these services according to their non-functional specifications, such as price, reliability and etc. In [ZBD+03], the authors propose methods for selecting desired Web services according to their attributes. The proposed architecture is based on two main classes, namely, coordinator classes and wrapper classes. Each service has a coordinator class that communicates with wrapper classes and other coordinator classes to achieve the execution of a certain workflow. The wrapper class, on the other hand, is responsible for receiving execution messages and translating them appropriately, suitable for being executed by the Web service. Once the execution is carried out, notifications are sent to the corresponding coordinator and the execution carries on. This model is meant to be a peer-to-peer, decentralized architecture; however, it is not quite evident how the coordination is supposed to happen without a priori knowledge of addresses of required coordinator classes and also without knowledge about the scenarios within which these classes should take part. It seems to us while this model does not require any glue code for execution of composite Web services, but it is not a loosely coupled, peer-to-peer model of execution either; this is due to the fact that each coordinator class has to have some knowledge about the general execution plan. Furthermore, for different execution plans, there should exist different wrapper and coordinator classes.

Each state of a state chart, representing a Web service, has some preconditions and postprocessing. Preconditions specify the logical conditions necessary to enter a state and postprocessings represent the actions that have happened as we exit a state. Postprocessings, in fact, determine the next state that the execution should continue in. Authors show how preconditions and postprocessings for a state can be computed by traversing through a compound transition; however, they fail to provide any logical reasoning tool capable of automatically computing these paths.

It seems to us this approach can be useful when the composition of services has been

\footnote{A set of states and transitions representing a possible execution path of a complex state.}
achieved and a model for execution of such a composite Web service is required; authors also emphasize more on suitability of this model for executing composite Web services. Although state charts’ formalism avoids the state explosion problem, usually occurred in these kinds of models; but it is often required to unfold the concise state chart representation of Web services into their compound transitions or their graph counter parts. This essentially amounts to the same problem as state explosion; see [BDSN02, ZBD+03] for how and why it is required to deal with more precise representation of Web services which would trigger the state explosion. Last, but not least issue with this model, is the fact that there is no specific methodology for composing Web services and the main focus is on how to represent the composite Web services rather than to define methods on how to discover and compose complex Web services.

7.1.4 A Semi-Automatic Approach

In [SHP03], the authors introduce one of the few approaches where semantic discovery of Web services is employed. The main goal of this system is to provide the user of a Web services composition system with the ability to browse through potentially helpful Web services and find and select his/her desired Web services. It is assumed that there exists a workflow of collaborating Web services, specified through some formalities, and the user is supposed to specify Web services capable of satisfying different parts of that workflow. The selection of Web services happens step by step. At each step, the output of a Web service is compared against the input of existing Web services in a knowledge base. The level of matching between the output messages and the input messages are evaluated by semantic methods introduced in [PKPS02].

This approach is both simple and pragmatic, however, it relies on some assumptions that makes its application rather limited. Firstly, it is required that the user of such a system to precisely specify the workflow and message passing scenarios of a composite Web service. This is a rather unrealistic assumption since often users do not have a clear vision of the composite service. Secondly, similar to many other approaches, there is no composition semantics and methodology for automating the composition of Web service.
7.2 Composition Algorithm

In this section we propose a Web service composition algorithm that receives a request for a Web service in form of an EPD and either returns a PIA as the response or informs that the request can not be satisfied. Our composition algorithm is based on interface models and the satisfiability relation that we have defined in previous chapters. We use the Web services satisfaction, described in Section 6.4, for checking the satisfiability. Moreover, we assume that we are able to use semantic matching features, described in Section 6.5, in this algorithm.

The repository of available Web services are assumed to be a local database storing the PIA representation of Web services which are specified in OWL-S. However, authors in [PKPS02] suggest a framework for associating OWL-S specification of Web services to their UDDI advertisement and in presence of such an architecture it is possible to use UDDI repositories and UDDI APIs (see Section 4.4) for finding a subset of relevant Web services and then run the algorithm over such services only. In either case, our assumption is that we are working on a repository of Web services in PIA presentation with their initial specification being in OWL-S.

We claim that given a certain request in form of an EPD and a repository of Web services in forms of PIAs, our algorithm can exhaust all possibilities of composition effectively and determine whether there exists any Web service or combination of Web services that satisfy such an EPD.

7.2.1 Decomposition of Requests

A certain request can be decomposed into multiple sub-requests in such a way that the composition of such sub-requests amount to the initial request. We believe that often the dependency rules in which users are interested in, suggest a natural decomposition. We can decompose such complex requests into some simpler requests and then carry out the satisfiability searches in a parallel manner. At the end the satisfying Web services can be assembled and checked for satisfiability against the main request.

Note that composition relations in different classes of interfaces often specify the scenarios for concurrent execution of their involved interfaces; sometimes, it is desired to specify
relations such as sequential executions of two interfaces or choice of execution among two interfaces. Composition is not capable of such specifications. Recall concatenation and choice operations in PIAs and EPDs, they specify the sequential and choice execution of two interfaces respectively. In fact, such operations are intuitive structures for decomposition of complex requests.

Any decomposition of a request to two EPDs, through either concatenation or choice operations, should be dealt with differently from decomposition through composition operation. As an example suppose that an EPD request $E$ can be decomposed to $E_1 \parallel E_2$ and $E_1$ can itself be decomposed to $E_{11} \land E_{12}$,\(^4\) thus:

$$E = (E_{11} \land E_{12}) \parallel E_2$$

Now suppose there exists $P_{11}$, $P_{12}$ and $P_2$ such that $P_{11} \triangleright_{\text{(PIA→EPD)}} E_{11}$, $P_{12} \triangleright_{\text{(PIA→EPD)}} E_{12}$ and $P_2 \triangleright_{\text{(PIA→EPD)}} E_2$. It is easy to show that $(P_{11} \land P_{12}) \triangleright_{\text{(PIA→EPD)}} (E_{11} \land E_{12})$.\(^5\) However, notice that commutativity and associativity, which PIA enjoyed on composition operation, does not hold in presence of concatenation and choice operation. That is to say, for example:

$$(P_{11} \land P_{12}) \parallel P_2 \neq P_{11} \land (P_{12} \parallel P_2)$$

Still, however, satisfiability checks can be done in a concurrent manner. That is $E_{11}$, $E_{12}$ and $E_2$ can be checked for satisfiability in a concurrent fashion but once the satisfying PIAs are recognized, they can not be assembled in an entirely concurrent manner. In fact, we require that explicit parenthesizations to be specified for all concatenation and choice operations in EPDs; and as such their satisfying PIA should follow similar parenthesizations. Composition of involved PIAs can then be only computed once the results of PIAs’ parenthesizations are computed.

As an example, consider that a user would like to search for a service that accepts a product name or a product id and then returns back the price of that product. Once the price of the product is retrieved, it is desired to make a payment by a credit card service. We decompose such a request into three separate EPDs, as shown in Figure 7.1. $E_{11}$ and

\(^4\)Recall that $\land$ represents the choice workflow pattern between two interfaces.

\(^5\)Furthermore, it can be shown that concatenation operation has similar properties. That is, if $P_{11} \triangleright_{\text{(PIA→EPD)}} E_{11}$ and $P_{12} \triangleright_{\text{(PIA→EPD)}} E_{12}$ then $(P_{11}P_{12}) \triangleright_{\text{(PIA→EPD)}} (E_{11}E_{12})$.
$E_{12}$ are related to each other through choice and $E_2$ can be composed with the result of such a choice. $E_{11}$ and $E_{12}$ each has a dependency rule while $E_2$ has two dependency rules $\langle (amount, amount), (credit\_no, ref\_no) \rangle$ and $\langle (amount, amount), (credit\_no, err\_no) \rangle$.

Notice that $E_2$ can conceptually be run sequentially after $(E_{11} \& E_{12})$ but since there is a shared port between $(E_{11} \& E_{12})$ output ports and $E_2$ input ports, they should technically be composed rather than concatenated.

Figure 7.2 represents the EPD which is the result of combining and composing the EPDs in Figure 7.1.

We believe that the decomposition of requests is an important issue and could significantly affect the performance of any Web services discovery system. On the other hand, wrong decompositions may lead to not being able to satisfying a request while a proper decomposition could have lead to satisfaction. Further investigation of this area is beyond the scope of this thesis.

### 7.2.2 Preprocessing of the Requests

Before submitting a request to the composition algorithm for finding a proper match for it, we can inspect whether it is necessary to decompose such a request for satisfying such a request. In this subsection, we show a situation in which we can automatically decompose a request to multiple requests connected to each other through the choice operation. The decomposed requests can then be satisfied while the original request, as a whole, may initially seem unsatisfiable.

Each dependency rule in an EPD represents a certain usage scenario of an EPD. Technically, we can decompose a certain EPD $E_1$ with two dependency rules to two EPDs, $E_{11}$ and $E_{12}$, each including only one of the dependency rules. Furthermore, $E_{11}$ and $E_{12}$ are required to include only those ports of $E_1$ that are used in their corresponding dependency rules. This simple decomposition is valid, assuming that all $E_1$’s ports are involved in at least one port dependency pair. Having this decomposition scheme in hand, we now introduce a scenario in which such a decomposition can lead to successful satisfaction while a complete request is not satisfiable.
Figure 7.1: A set of requests connected through choice and composition operations. The dashed box around $E_{11}$ and $E_{12}$ represents the result of $E_{11} \land E_{12}$. Notice that the result’s output port, price, is connected to $E_2$’s input port, amount, through a channel.
Consider Figure 7.1 and EPD $E_1 = \langle I_{E_1}, O_{E_1}, O_{E_1}^+, H_{E_1}, V_{E_1} \rangle$ where:

- $I_{E_1} = \{ prod\_id, prod\_name \}$
- $O_{E_1} = \{ price \}$
- $O_{E_1}^+ = \{ price \}$
- $H_{E_1} = \emptyset$
- $V_{E_1} = \{ v_{E_1}^1, v_{E_1}^2 \}$
- $v_{E_1}^1 = \langle (prod\_id, price) \rangle$
- $v_{E_1}^2 = \langle (prod\_name, price) \rangle$

$E_1$ is vulnerable for decomposition if there does not exist any PIA, $P_1$, such that $P_1 \downarrow_{(PIA\rightarrow EPD)}^\text{Strong} E_1$. If such a $P_1$ does not exist then there could not exist any composition of some PIAs such that they satisfy $E_1$. This is since any $P_{11}$ and $P_{12}$, where each of them satisfies only one of the dependency rules $v_{E_1}^1$ and $v_{E_1}^2$, respectively, are not composable. This problem basically arises from the fact that composition in PIAs can not express the \textit{choice} operation.

We generalize this situation and apply it to the situations where some dependency rules belonging to an EPD have some ports in common and that EPD can not be satisfied immediately. In our example, $v_{E_1}^1$ and $v_{E_1}^2$ have the port \textit{price} in common and thus we can divide $E_1$ into $E_{11} \land E_{12}$ as illustrated in Figure 7.1. As a preprocessing procedure before submitting a request to the composition algorithm, we first check whether there exists a single Web service satisfying that request. If such a Web service does not exist then
we inspect the dependency rules of that EPD and check whether they have any common ports. If there exists some common ports then we should either be able to satisfy any such group of dependencies by a single Web service or we have to decompose that EPD into some sub-requests connected to each other through choice control construct. Each of such sub-requests consist of only one of the dependency rules and are more likely to be satisfied individually where as the whole requests, as shown in the above example, is not satisfiable at all.

7.2.3 Algorithm

So far we have been only involved in situations where given a request, as an EPD, and a Web service specification, specified as a PIA, we were able to inspect whether a certain satisfiability relation holds. Obviously, this is not always the case; consider the situation where a number of PIAs can be composed to satisfy a certain request. In this section, we introduce an algorithm that receives a request and checks for the existence of a match satisfying that request; if such a match does not exist then this algorithm begins with the PIA that “most probably” can satisfy that request, or part of the request. If a part of the request is satisfied, then the algorithm continues with searching for PIAs that by composing them with the partially satisfying PIAs, we can achieve complete satisfaction or perhaps proceed an incremental step towards a more satisfying state. This incremental approach is modelled through recursive function calls to the algorithm. Each time a smaller request is being passed and the whole request is satisfied if all of the recursive function calls succeed.

This algorithm is looking for PIAs that most probably can satisfy a request. We notice that our first choice for the most probable PIA may not be the best choice. In other words, it is quite possible that a composition of a set of PIAs that are not most probably the best candidates for satisfaction of a request, manage to satisfy the original request. Thus, we should have means to inspect all possible compositions that may satisfy the request. In other words, if necessary, the algorithm should be able to exhaustively check for all possible satisfying compositions.

In situations where there does not exist a single PIA satisfying a request, the problem reduces to: “composition of which PIAs most probably can satisfy that request?”

The criteria we choose in our algorithm for the best chance of satisfiability is maximum
7.2. COMPOSITION ALGORITHM

IO similarity between a request and a Web service. We believe that the more similarity exists between inputs and outputs of a request and inputs and outputs of a Web service, the more likely it would be that the Web service is helpful in satisfying the request. In the following definition we make these criteria concrete. We first specify a notation for the set of PIAs which have IO similarity with a certain EPD and then introduce a notation for specifying PIAs which can consume extra unnecessary IO of partially helpful PIAs.

Definition 7.1 Given an EPD \( F = \langle I_F, O_F, O_F^+, H_F, V_F \rangle \) and PIA \( P = \langle V_{abst}^P, V_{init}^P, V_{comp}^P, A_{I}^{P,simp}, A_{O}^{P,simp}, A_{H}^{P,simp}, A_{P}^{comp}, \tau_{abst}^P, \phi_p \rangle \), \( P \) has IO similarity with \( F \) if the following conditions are true:

- \( ((A_{I}^{P,simp} \cup A_{H}^{P,simp}) \cap O_F = \emptyset) \land ((A_{O}^{P,simp} \cup A_{H}^{P,simp}) \cap I_F = \emptyset) \)
- \( (A_{I}^{P,simp} \cap I_F \neq \emptyset) \lor (A_{O}^{P,simp} \cap O_F \neq \emptyset) \)

The above comparison between ports of EPDs and actions in PIAs are carried out as described in semantic message comparison in Section 6.5. In other words, any valid level of semantic matching, i.e. exact, plug in, subsumes, would result in a positive comparison. Unlike the semantic matching proposed in Section 6.5, IO similarities does not require that all messages to be satisfied. In fact, an automaton with a single IO similarity with the request EPD, can be considered having IO similarities with the request.

Having two PIAs \( P_1 \) and \( P_2 \), that have IO similarity with \( F \), then \( P_1 \) has more IO similarities with \( F \) than \( P_2 \) if one of the following is true:

- \( (\text{count}(A_{P_1}^{simp} \cap P_F^+) > \text{count}(A_{P_2}^{simp} \cap P_F^+)) \)
- \( (\text{count}(A_{P_1}^{simp} \cap P_F^+) = \text{count}(A_{P_2}^{simp} \cap P_F^+) \land \text{count}(A_{P_1}^{O,simp} \cap O_F) > \text{count}(A_{P_2}^{O,simp} \cap O_F)) \)

The above similarity criteria is a quantitative criteria. In the case of two PIAs having the same number of similar IO with an EPD, the PIA with more output similarity has a better overall IO similarity. In this way, the higher IO similarity there is, the more helpful the corresponding PIA is considered to be.
Definition 7.2 Given an EPD $F = \langle I_F, O_F, O_F^+, H_F, V_F \rangle$, the set $\text{IOSimilar}(F) = \langle P_1, P_2, \ldots, P_n \rangle$ is the sequence of PIAs that have some level of IO similarity with $F$. Furthermore, for all $P_i$ where $0 \leq i < n$, $P_{i+1}$ has an equal or higher level of similarity with $F$ than $P_i$ does.

Having a PIA, $P$, which has partial similarity with EPD $F$, there exists a scenario where we can compose $P$ with $P'$ so that some of the extra IO of $P$ are consumed by $P'$, and moreover some extra IO, not present in $P$, would show up which can lead us to a complete satisfaction. To make this scenario more concrete, as an example,\(^6\) consider a request for a “French-Persian” dictionary Web service. On the other hand, assume that in our repository of Web services we have “French-English”, “English-German” and “German-Persian” dictionaries. Clearly, by composing these three Web services we can have “French-Persian” dictionary. To realize that such a composition is useful, we should first realize that starting from “French-English” dictionary we should continue with “English-German” and then “German-Persian” dictionary compositions. Our proposed algorithm is capable of recognizing and carrying out such chains of compositions.

Our proposed algorithm actually starts by composing “French-English” and “German-Persian” services first and then finally compose them with “English-German”. These compositions seem irrelevant from IO similarity point of view but still they are useful to create the desired service. Essentially, these scenarios would happen when we are dealing with such compositions where the outputs of a service is required to be directed to the input of another service in a pipe-lined manner. To make our algorithm to support such a paradigm, we should let our algorithm to use such intermediary services whenever necessary. The following definition specifies Web services that can be used in compositions, as intermediary, to eventually satisfy requests.

Definition 7.3 Given an EPD $F = \langle I_F, O_F, O_F^+, H_F, V_F \rangle$, and PIA, $P = \langle V_P^{\text{abst}}, V_P^{\text{init}}, V_P^{\text{comp}}, A_P^{I_{\text{simp}}}, A_P^{O_{\text{simp}}}, A_P^{H_{\text{simp}}}, A_P^{\text{comp}}, \tau_P^{\text{abst}}, \phi_P \rangle$, where $P$ has IO similarities with $F$, we define the set $\text{IOConsumer}(F, P) = \langle P'_1, P'_2, \ldots, P'_m \rangle$ as a sequence of PIAs. where each

---

\(^6\)The idea of this example is due to the motivating example in [SHP03].
7.2. COMPOSITION ALGORITHM

\( P'_i \in IOConsumer(F, P) \) is composable with \( P \) and the following is true:

\[ (A^{simp}_P \setminus P^+_F) \cap A^{simp}_{P'_i} \neq \emptyset \]

Furthermore, for each \( P'_i, P'_j \in IOConsumer(F, P) \), where \( i < j \), one of the following conditions is true:

- \( \text{count}((A^{simp}_P \setminus P^+_F) \cap A^{simp}_{P'_i}) > \text{count}((A^{simp}_P \setminus P^+_F) \cap A^{simp}_{P'_j}) \)
  
  Meaning that \( P'_i \) is capable of consuming more of the \( P \)'s extra useless input and output action than \( P'_j \) can do.

- \( \text{count}((A^{simp}_P \setminus P^+_F) \cap A^{simp}_{P'_i}) = \text{count}((A^{simp}_P \setminus P^+_F) \cap A^{simp}_{P'_j}) \land \text{count}((A^{simp}_P \setminus P^+_F) \cap A^{inp}_{P'_i}) > \text{count}((A^{simp}_P \setminus P^+_F) \cap A^{inp}_{P'_j}) \)

  In case \( P'_i \) and \( P'_j \) are consuming the same number of extra useless input and output actions of \( P \) then the one that consumes more extra input actions of \( P \) is considered more helpful.

\[ \blacksquare \]

Before presenting our algorithm we introduce one further definition which enables us to remove some of the input and output ports of a certain EPD and define a new EPD not including those input and output ports.

**Definition 7.4** Given an EPD \( F = \langle I_F, O_F, O^+_F, H_F, V_F \rangle \) and a set of ports \( t \subseteq P^+_F \), we define the operation remove ports on \( F \) and \( t \), \( F \circ \text{ports} \), as a binary operation resulting a new EPD \( F' = \langle I_{F'}, O_{F'}, O^+_F, H_{F'}, V_{F'} \rangle \) defined as following:

- \( I_{F'} = I_F \setminus t \)
- \( O^+_{F'} = O^+_F \setminus t \)
- \( O_{F'} = O_F \setminus t \)
For each $\nu \in \mathcal{V}_F$ where $\nu = \langle (a_1, b_1), (a_2, b_2), \ldots, (a_n, b_n) \rangle$ we create $\mu \in \mathcal{V}_{F'}$ such that:

- $\mu \sqsubseteq \nu$
- $\forall (a, b) \in (\nu \ominus \mu) \Rightarrow (a \in t) \lor (b \in t)$

Having defined the necessary notations, we are ready to define our composition algorithm.

The algorithm in Figure 7.3 is a recursive algorithm that works on a repository of existing Web services, which are all presented as their PIA equivalents. We require that the following assumptions are satisfied for such a repository:

1. Each PIA representation of a Web service, in the repository, is linked to its OWL-S specification. Hence, it is possible to investigate the semantic matching of inputs and outputs. Moreover, any matching result can be actually executed through its reference to its corresponding OWL-S document(s), and consequently accessing the necessary WSDL documents.

2. Web services advertised in such a repository are stored as Web services without any explicit loop structures. That is, PIA corresponding to each Web service can be mapped to an acyclic graph. We require this criteria since our satisfiability algorithm relies on acyclic graphs. This is not a restrictive assumption, it is easy to make any Web service to work in a loop structure, as shown in Section 4.7.

3. The repository should be equipped with necessary index structures enabling us to retrieve IO similar and IO consumer PIAs efficiently. In fact, these are the two major sets of operations on the repository of PIAs that can impose significant costs to the algorithm. This becomes more crucial when such a repository contains relatively large number of PIAs. However, we believe that by having different sets of repositories, based on different types of businesses, and by appropriately indexing such repositories, it is possible to efficiently search on such repositories.
Algorithm Boolean COMPOSITION(EPD F, PIA P);
Result : P
Variables: i = 0, Old_F = ∅, X = ∅;
beg

if F ≠ ∅ then
P = ∅;
return True;
endif

X = IOSimilar(F) ⊕ IOConsumer(F, P);
for i = 1...count(X) do

Old_F = F;
F = F ⊕ A_{X_i}^{simp};
if ! COMPOSITION(F, P) then
F = Old_F ;
else
if P ≠ ∅ then
P = P || X_i;
else
P = X_i;
endif
endif
if P ▶_{(PIA→EPD)}^{Strong} Old_F then
return True;
else
F = Old_F ;
endif
endfor
return False;
end

Figure 7.3: The outline of the algorithm for finding satisfying Web service for request F. The return value of the algorithm would be a True or False value. If the return value is True the satisfying Web service can be found in P; where P represents the PIA satisfying that Web service.
7.2.4 Algorithm Description

This algorithm recursively composes helpful Web services to satisfy a certain request. The criteria for helpfulness is primarily IO similarity and then the capability to consume extra inputs and outputs of compositions. In each recurrence of the algorithm, we receive the latest composition that the algorithm has produced thus far. In this way, the algorithm incrementally tries to reach the point where all dependency rules are satisfied. Throughout this process, all possible compositions that can be helpful are considered, however, we believe that IO similarity and IO consumption criteria are helpful to achieve the target composition soon.

In the first recurrence of the algorithm, the request $F$ along with an empty $P$ is passed to the composition algorithm, COMPOSITION. If the request is empty then clearly the result would also be an empty service. Notice that the empty condition serves as the base case of the recurrence and once we reach this stage, it means that we can return back to continue other recurrences of the algorithm. In the case where $F$ is not empty then we create a sequence of PIAs in $X$ that can potentially be helpful in satisfying $F$. As mentioned earlier, our criteria for helpfulness is based on the IO similarities between PIAs and $F$. Furthermore, we compute all PIAs that can consume extra unnecessary IO of $P$ by computing $IOConsumer(F, P)$ and appending them after PIAs with IO similarities.

As you can see, this algorithm considers IO similar PIAs more helpful than IO consumer PIAs.

Having created the sequence $X$, we inspect all members of $X$ one by one; starting from the first member, suppose it is $x_1$, we first adjust $F$ by removing those ports belonging to $x_1$, $(F \ominus x_1)$, and then trigger another recurrence of the algorithm. The second recurrence again checks whether $F$ is empty, if it is, it returns True which would let the first recurrence to continue. In fact, at this point we know that $x_1$ includes all input and output messages necessary to satisfy $F$ and we should just check whether the satisfiability relation, $\Rightarrow_{Strong}(PIA \rightarrow EPD)$, actually holds. If it holds then we know that $x_1$ is our match and we return it back. If it does not then we continue by trying $x_2$. However the second recurrence of the algorithm may not return True which means that $x_1$ does not contain all necessary input and output messages for satisfying $F$. In that case, the second recurrence would act just like the first recurrence and would create its own $X$ list and tries to find a match for the
reduced request.

The recursions continue until we reach the empty request. All partially satisfying PIAs are composed with each other whenever a recursion comes back. At the end the result of all PIAs’ compositions is checked for satisfiability. In this way, we do not check for refinement unless we are sure a certain automaton includes all the necessary input and output actions.

Given a request, if there exists any composition, using PIAs in the repository, that satisfies that request, this algorithm will find it. This can be obviously observed by the fact that, if necessary, this algorithm exhaustively inspects all possibly helpful compositions. However, we believe the concept of IO similarity helps us to find such a composition early.

The complexity of the execution for this algorithm heavily depends on the methods that we choose for indexing Web services repository, which is beyond the context of this thesis. Assuming that there are $N$ PIAs that have IO similarity with request $F$, without considering the cost of retrieving the necessary Web services from the repository, exponential time in $N$ is required for inspecting all possible compositions of $N$ PIAs. Notice that since composition in PIAs is commutative and associative, the number of possible compositions is independent of the order of the composition. Also, note that exponential time is a rough higher bound estimate; we expect the IO similarity to be an effective criteria for finding the matches in early stages of the algorithm.

### 7.3 Glue Code Generation

Once we have a PIA satisfying our request, we can follow that automaton’s links to appropriate OWL-S documents. Having the necessary OWL-S documents in hand, by inspecting the grounding classes of those OWL-S documents we can access the corresponding WSDL documents which actually enable us to execute the corresponding services. Thus, having the satisfying automaton in hand, it is easy to run the service. On the other hand, recall that PIAs require a helpful environment to run. Such an environment basically provides the necessary inputs and accepts all of outputs in different states. It is easy to create such

\(^7\)Notice that some other approaches introduce exponential order of execution on the size of the repository. Furthermore, not all of those approaches are capable of performing their tasks in concurrent fashions, as we propose.
an environment automaton which that such inputs and accept such outputs.

An environment for a PIA can be considered the template of the glue code for executing underlying Web services, which such automaton is modelling. As an example, suppose that the PIA in Figure 3.12, in page 82, is retrieved as the result for a certain request. It is easy to create a PIA, as shown in Figure 7.4, that provides all necessary inputs and accepts all outputs in appropriate states. The algorithm for creating such an automaton is quite straightforward. It starts from the initial state of the automaton and whenever there is an input action it adds an output action to the environment automaton. Conversely, whenever there is an output action, an input action is added to the environment automaton.

The above procedure can only provide the template for the glue code. In fact, it only provides the possible sequences of message passing between the environment and the service. In presence of different input actions, enabled at one state, there should exist a decision making mechanism that would choose to issue the appropriate output message. This decision making can be referred to as the business logic of the glue code. However, it is efficient to provide a draft glue code and let the users only be responsible for completing such a draft.
7.4 Advantages of This Approach

Our approach for Web service discovery and composition has some explicit advantages over other approaches. Below, we mention such advantages:

1. To the best of our knowledge, we are the first to propose an approach in which different formalisms are used for capturing user requests, i.e. using EPD, and modelling the Web services, i.e. using PIA. We believe that it is crucial to consider different levels of abstraction for these two requirements.

2. We present a comprehensive and coherent set of formalisms capable of modelling different aspects of a Web services discovery system, i.e., request capturing, Web services modelling and formal correlations between requests and the services.

3. We provide intuitive operations for assembling different PIAs together, i.e. composition, concatenation and choice operations. Our composition operation is consistent with the inherent properties of Web services which is based on a synchronized and message passing paradigm of computation. Other approaches often fail to provide any composition methods for assembling different Web services. Even those approaches providing composition operations usually rely on some ad hoc mechanisms incapable of expressing different composition scenarios.

4. Using EPDs as the model for capturing the user requests, we can specify the requests as a set of EPDs being connected to each other through composition. This is a nice separation, which enables us to systematically decompose a certain request.

5. Having a certain request decomposed and using the satisfiability relation between PIAs and EPDs, we are capable of concurrently checking for satisfiability relations and later check, if necessary, whether the satisfiability holds for their composition.

6. We have laid the foundation of our approach on top of semantic Web standards and thus are capable of reasoning about semantic matching of input and output elements of the system.

7. Having discovered a matching Web service, it is possible to automatically create a draft version of the glue code that can execute that service.
Chapter 8

Conclusions and Future Work

In this chapter we present the conclusions of this thesis; we also present, in Section 8.2, our future plans for pursuing and investigating further the ideas we have presented in this thesis.

8.1 Summary and Conclusions

We extend the concept of interface models, by introducing some new interfaces, and apply those models to some software engineering problems. The models we propose, provide the level of expressivity with which we can simulate different software artifacts.

Following, we briefly describe how each of our proposed models enhance their predecessors. In general, our models are different from other interface models proposed so far, in the sense that they only aim at modelling software artifacts rather than providing general-purpose interface models dealing with software/hardware systems.

- **Stateless Multiple Port Dependency Interface (MPD):** While stateless interfaces, at first, do not seem useful models in the context of software systems, since software artifacts are usually stateful artifacts; we find them helpful in modelling user requests in software matching systems. An outside viewer/requester of a software artifact often do not have a clear notion of the states in which that software operates in; instead, they usually know what sorts of input and output are expected
of such systems. Stateless interface models provide a certain level of abstraction that is suitable for modelling such requests.

MPD is loosely based on PD. However, connections and dependencies in MPDs are different from its predecessor. In PD each interface instance has its own connection while in MPDs a certain connection can be shared among all involved interface instances that are connected through that connection. Furthermore, a connection can not connect the output ports of a certain interface to input ports of the same interface, this limitation intuitively makes sense in the context of software systems, where as in hardware systems such connections are common.

In general, although MPD is based on PD, MPD holds a different set of properties which makes it a useful model for modelling some aspects of software systems.

• **Enhanced Stateless Port Dependency Interface (EPD):** EPDs are an attempt to create an interface model as concise as the stateless interface models with some expressivity capability of stateful models (especially in modelling temporal order of execution). When we tried to bridge between stateless and stateful worlds, we realized that interface well-formedness criteria are not flexible enough for such a bridging. In fact, it turned out that those criteria either lead to completely stateful models or completely stateless models. To achieve such an intermediary model, we made the compromise of eliminating composition associativity from the well-formedness criteria. We call such interface models weak interfaces. In the context which we use EPDs, weak interfaces are as helpful as a similar model enabled with associativity.

An observation, worth noting, in this attempt is the elegance which commutativity along with associativity provide for any binary operation. These two properties combined together provide the chance for performing a certain operation in a concurrent manner. We came to appreciate this useful property while attempting to make EPD composition associative, that turned out to be infeasible.

• **Parameterized Interface Automata (PIAs):** PIAs are stateful interfaces capable of modelling communicative software artifacts. Artifacts could be in form of components, services, objects and etc. PIAs are different from IAs in the sense that they provide a higher level of expressivity necessary for specifying the parameters of
input and output actions. Having this capability, they are more precise and their composition tend to be smaller in size, only capturing possible scenario of executions rather than considering all possible interleaving of states belonging to the involved artifacts.

While our proposed interface models, by themselves, are useful modelling formalisms, we realize that different aspects of a certain system can be modelled using different formalisms. Thus, it is helpful to study the correlation of different interface formalisms. We propose satisfiability relations and specify some well-formedness criteria for any valid satisfiability relation. Interface models support top-down design and thus each interface can be decomposed to several interface models. As such, satisfiability relation can be carried out for each element of a decomposition in a concurrent manner and ultimately check whether composition of multiple satisfying interfaces satisfies the original request.

We applied our proposed set of formalism to the problem of Web services discovery and composition. PIAs and EPDs tend to be suitable models for simulating Web services behaviors and user requests, respectively. We introduced two satisfiability relations between PIAs and EPDs and showed how such relations combined with semantic matching techniques can be used in the problem of Web services discovery. We also provide an efficient algorithm for Strong satisfiability relation between EPDs and PIAs.

Decomposition of a search request turned out to be an important issue in the context of our approach to Web service discovery. Some of unsatisfiable search requests can be decomposed such that their constituent sub-requests become satisfiable. We proposed some of the obvious decomposition in this respect, which can be easily recognized in search requests.

On the other hand, some search requests can not be satisfied by exactly one Web service. However, composition of a set of Web services can satisfy such a request. We propose an algorithm which effectively exhausts possible Web services compositions searching for helpful compositions.
8.2 Future Work

While working on this thesis, we recognized some research topics that we are interested to further pursue. The direction of such researches can be categorized as following:

- **Behavioral Matching of Web Services:** In this thesis we only presented methods for non-behavioral matching of Web services. However, as mentioned earlier, non-behavioral matchings do not necessarily guarantee that the matching software artifacts represent semantically similar functionalities. To guarantee the perfect matching we have to check semantic similarities of matching software artifacts too. In the context of Web services, we are interested in using situation calculus [Rei01] to achieve such a vision.

  There are promising clues that suggest that situation calculus can complement PIA representation of Web services effectively. Both formalisms are built on top of two basic concepts: *actions* and *states* (in situation calculus the term is *situation*). Furthermore, both formalisms rely on actions for any state (or situations) transitions. Intuitively, there is a nice correlations between different concepts in the two formalisms which suggest that they can be used in parallel to achieve a comprehensive matching mechanism.

- **Interface Decomposition:** Interfaces are meant to support top-down design and decomposition. Manual decomposition, however, can be quite a subjective matter. The problem of decomposing the user requests into some sub-requests is of great importance. In this thesis, we briefly described the obvious decomposition which are mandatory in the process of Web service discovery/composition. We believe this problem can be further investigated in this context. Any smart decomposition of such requests can increase the performance of such systems drastically.

- **Smarter Composition Algorithms:** Our Web service composition approach mainly relies on IO similarity criteria for the priority of compositions. In presence of semantic specification of Web services, we can take advantage of those information in conjunction with IO similarity criteria. Furthermore, we are looking for more specific and
efficient criteria, in addition to IO similarity, for carrying out the compositions and thus avoiding some of the unnecessary compositions.

- **Structure of Web Services Repositories:** Among different Web services specification standards, only UDDI has a well-defined structure for storing and retrieving information about Web services specification. For semantic specification of Web service, there is no specific standards for such repositories. We are interested in investigating properties of efficient repositories/databases for storing semantically enabled specification of Web services. In particular, issues such as using distributed vs centralized repositories and efficient mechanisms for updating and searching such repositories are of our interests.
Appendix A

Proofs

In this appendix we provide the proofs and proof sketches for the theorems and lemmas proposed in chapters 2 and 3.

A.1 Chapter 2 Proofs

**Theorem 2.1** For all MPDs $F$ and $G$, either they are not composable or $F \parallel G$ is defined and is equal to $G \parallel F$.

Proof. Observe that the composition operation in MPDs, as defined in Definition 2.5, is completely symmetric with respect to the involving MPDs and thus is commutative. □

**Theorem 2.2** For all MPDs $F$, $G$ and $K$, either some of the MPDs are not composable or $(F \parallel G) \parallel K = F \parallel (G \parallel K)$.

Proof Sketch. It can be observed that to have $(F \parallel G) \parallel K$ defined, it is necessary that sets of shared ports $\text{SharedPorts}(F, G)$, $\text{SharedPorts}(F, K)$ and $\text{SharedPorts}(G, K)$ should be disjoint.¹ Having this fact in mind, in the following, we consider five elements of MPDs and show how associativity is valid in those contexts.

¹Otherwise some of the MPDs will not be composable
• We start by showing that \( I_{(F \| G)\| K} = I_{F\| (G\| K)} \). By the composition definition we have:

\[
I_{(F \| G)\| K} = (((F \cup G) \setminus \text{SharedPorts}(F, G)) \cup I_K) \setminus \text{SharedPorts}((F \| G), K)
\]

Notice that \( \text{SharedPorts}(F, G) \) and \( I_K \) are disjoint sets and thus we can write the above equation as below:

\[
I_{(F \| G)\| K} = (((I_F \cup I_G \cup I_K) \setminus \text{SharedPorts}(F, G))) \setminus \text{SharedPorts}((F \| G), K)
\]

Which can be written as following:

\[
I_{(F \| G)\| K} = ((I_F \cup I_G \cup I_K) \setminus (\text{SharedPorts}(F, G) \cup \text{SharedPorts}((F \| G), K) \cup \text{SharedPorts}(G, K)))
\]

(A.1)

On the other hand, we can substitute shared ports with their real values as following:

\[
\text{SharedPorts}(F, G) = P^+_F \cap P^+_G \\
\text{SharedPorts}((F \| G), K) = (P^+_F \cup P^+_G) \cap P^+_K
\]

(A.2)

By doing the proper substitution of values in Equation A.2 in Equation A.1 and expanding the formulas, we will have the following:

\[
I_{(F \| G)\| K} = (((I_F \cup I_G \cup I_K) \setminus (\text{SharedPorts}(F, G) \cup \text{SharedPorts}(F, K) \cup \text{SharedPorts}(G, K)))
\]

(A.3)

Similarly, \( I_{F\| (G\| K)} \) can be shown to be equal to Equation A.3 which implies \( I_{(F \| G)\| K} = I_{F\| (G\| K)} \).

• Similar for the set of input ports, it is possible to show that \( O_{(F\| G)\| K} = O_{F\| (G\| K)} \).

• Now, we show that \( O^+_{(F\| G)\| K} = O^+_{F\| (G\| K)} \). By the composition definition we have:

\[
O^+_{(F\| G)\| K} = (O^+_{(F\| G)} \cup O^+_K) \cup \text{SharedPorts}((F \| G), K)
\]

By substituting shared ports with values in Equation A.2, it is trivial to show that \( O^+_{(F\| G)\| K} = O^+_{F\| (G\| K)} \).

\[\text{It is trivial to show that given any three sets, } X, Y \text{ and } Z, (X \setminus Y) \setminus Z = X \setminus (Y \cup Z).\]
A.1. CHAPTER 2 PROOFS

• We should now show that \( H_{(F \parallel G) \parallel K} = H_{F \parallel (G \parallel K)} \). By the composition definition, we have:

\[
H_{(F \parallel G) \parallel K} = (H_F \cup H_G \cup \text{SharedPorts}(F, G)) \cup H_K \cup \text{SharedPorts}((F \parallel G), K)
\]

On the other hand, we have:

\[
\text{SharedPorts}((F \parallel G), K) = \text{SharedPorts}(F, K) \cup \text{SharedPorts}(G, K)
\]

Thus, we have:

\[
H_{(F \parallel G) \parallel K} = (H_F \cup H_G \cup H_K) \cup \text{SharedPorts}(F \parallel G) \cup \text{SharedPorts}(F \parallel K) \cup \text{SharedPorts}(G \parallel K)
\]

Similarly, it can be shown that \( H_{F \parallel (G \parallel K)} \) will amount to the above set and thus \( H_{(F \parallel G) \parallel K} = H_{F \parallel (G \parallel K)} \).

• Since the sets of shared ports between the three MPDs \( F, G \) and \( K \) are disjoint, we can show that for all sets \( u^i_F \in U_F, u^j_G \in U_G \) and \( u^k_K \in U_K \), the following is true.

\[
\left| u^{(i,j),k}_{(F \parallel G) \parallel K} \right| = \left| u^{i,(j,k)}_{F \parallel (G \parallel K)} \right|
\]

\( u^{(i,j),k}_{(F \parallel G) \parallel K} \) can be defined as following:

\[
\left| u^{(i,j),k}_{(F \parallel G) \parallel K} \right| = \left| (((u^i_F \cup u^j_G)^* \setminus A) \cup u^k_K)^* \right| \setminus B
\]

Where:

\[
A = (\text{SharedPorts}(F, G) \times P^+_F \parallel G) \cup (P^+_F \parallel G \times \text{SharedPorts}(F, G))
\]

\[
B = (\text{SharedPorts}((F \parallel G), K) \times P^+_G \parallel G, k) \cup (P^+_G \parallel G, k \times \text{SharedPorts}((F \parallel G), K))
\]

Notice that \( A \cap u_K^k = \emptyset \), this is since if there exist an element in \( A \cap u_K^k \), it implies that the set of shared ports between the three MPDs are not disjoint, which is not true. Thus, we have:

\[
\left| u^{(i,j),k}_{(F \parallel G) \parallel K} \right| = \left| (((u^i_F \cup u^j_G)^* \cup u^k_K)^* \setminus A)^* \right| \setminus B
\]
Observe that since \( A \cap u_k^k = \emptyset \), we can defer subtracting \( A \) from \(((u_F^i \cup u_G^j)^* \cup u_K^k)^*\) after computing that value. Members of \( A \) will not participate in creating any new member in \(((u_F^i \cup u_G^j)^* \cup u_K^k)^*\) because \( A \cap u_k^k = \emptyset \). Thus, subtracting \( A \) before or after the computation of \(((u_F^i \cup u_G^j)^* \cup u_K^k)^*\) will not change the result and we have:

\[
u^{(i,j),k}_{(F\parallel G)\parallel K} = (((((u_F^i \cup u_G^j)^* \cup u_K^k)^*) \setminus A) \setminus B
\]

which is equal to:

\[
u^{(i,j),k}_{(F\parallel G)\parallel K} = (((u_F^i \cup u_G^j)^* \cup u_K^k)^*) \setminus (A \cup B)
\]

It can also be proved that \(((u_F^i \cup u_G^j)^* \cup u_k^k)^*\) and \((u_F^i \cup u_G^j \cup u_k^k)^*\) yield the same value. This is because members belonging to \((u_F^i \cup u_G^j)^*\) computed through transitive closure can also be computed by \((u_F^i \cup u_G^j \cup u_k^k)^*\) and furthermore, \((u_F^i \cup u_G^j \cup u_k^k)^*\) can not introduce more members computed through transitivity with \((u_F^i \cup u_G^j)\) being involved. Thus, we have:

\[
u^{(i,j),k}_{(F\parallel G)\parallel K} = (((u_F^i \cup u_G^j \cup u_k^k)^*) \setminus (A \cup B)
\]

On the other hand, by substituting \( A \) and \( B \) with their corresponding values in Equation A.2, we have the following:

\[
u^{(i,j),k}_{(F\parallel G)\parallel K} = (((u_F^i \cup u_G^j \cup u_k^k)^*) \setminus ((P_F^+ \cap P_G^+ \cup P_K^+) \cup (P_F^+ \cap P_K^+ \cup P_K^+)))
\]

\[\times (P_F^+ \cup P_G^+ \cup P_K^+)] \tag{A.4}
\]

In a similar fashion, we can show that \( u^{(i,j,k)}_{(F\parallel G)\parallel K} \) would amount to the right hand side of Equation A.4. Thus, we conclude for any \( u^{(i,j),k}_{(F\parallel G)\parallel K} \), it is equal to \( u^{(i,j),k}_{(F\parallel G)\parallel K} \) and this concludes this part of our proof.

Having shown that all elements of \((F \parallel G) \parallel K\) are equal to \( F \parallel (G \parallel K) \), we conclude that they are equal.

**Theorem 2.3** *MPDs are interfaces.*

**Proof Sketch.** Having shown that composition in MPDs is both commutative and associative, we now show that MPD complies with two well-formedness criteria of interfaces defined in Subsection 2.1.1.
1. To prove the first condition of well-formedness for interfaces, we should show that given two composable MPDs \( F = \langle I_F, O_F, O^+_F, H_F, U_F \rangle \) and \( G = \langle I_G, O_G, O^+_G, H_G, U_G \rangle \), where \( U_F = \{ u^1_F, u^2_F, \ldots, u^n_F \} \) and \( U_G = \{ u^1_G, u^2_G, \ldots, u^m_G \} \) and MPD \( F' = \langle I_{F'}, O_{F'}, O^+_{F'}, H_{F'}, U_{F'} \rangle \) where \( U_{F'} = \{ u^1_{F'}, u^2_{F'}, \ldots, u^p_{F'} \} \), if \( F \parallel G \) is defined and \( F' \preceq F \) then \( F' \parallel G \) is also defined and \( F' \parallel G \preceq F \parallel G \).

To show that \( F' \parallel G \) is defined, we should show that it complies with three compositability criteria mentioned in Definition 2.4.

(a) Obviously \( F' \) and \( G \) use the same interconnect, \( \theta \). This is because \( F \) and \( G \) use the same connections and, on the other hand, \( F' \), as refinement of \( F \), uses the same connection as \( F \) uses. This is since \( O^+_{F'} \subseteq O^+_F \), \( O_F \subseteq O_{F'} \) and \( I_{F'} \subseteq I_F \) and thus, obviously, \( \text{SharedPorts}(F', G) \subseteq \text{SharedPorts}(F, G) \). This suggests that all members of \( \text{Shared}(F', G) \) participate in interconnect \( \theta \), as required in Definition 2.4.

(b) As the second requirement for compositability, it should be shown that \( H_{F'} \cap H_G = \emptyset \). This is obvious since by compositability of \( F \) and \( G \), we know that \( H_F \cap H_G = \emptyset \) and, on the other hand, \( H_{F'} \subseteq H_F \). Thus, \( H_{F'} \cap H_G = \emptyset \).

(c) It could be shown that dependency pairs belonging to any two arbitrary dependency sets belonging to \( F \) and \( G \), do not generate cycles of dependencies through transitivity. In general, this is true since there are less shared ports between \( F' \) and \( G \) than \( F \) and \( G \). Furthermore, \( F' \) dependencies do not impose more dependencies on \( F \) ports than \( F \) itself does. Thus, since \( F \) and \( G \) do not create such cycle of dependencies, \( F' \) and \( G \) can not create such cycles either.

Complying with compositability criteria of MPD, we conclude that \( F' \parallel G \) is defined.

Now we should show that, in fact \( (F' \parallel G) \preceq (F' \parallel G) \). To show that, we should prove that four refinement conditions, defined in Section 2.2.2, are respected.

(a) As the first condition, we should show that:

- \( I_{(F' \parallel G)} \subseteq I_{(F \parallel G)} \)
- \( O_{(F' \parallel G)}^+ \subseteq O_{(F \parallel G)} \)
• $O_{(F'\parallel G)} \supseteq O_{(F\parallel G)}$

It is trivial to prove the above by considering definition of composition in MPD and the following facts, which are implied by MPD refinement definition.

• $I_{F'} \subseteq I_F$
• $O^+_{F'} \subseteq O^+_F$
• $O_{F'} \supseteq O_F$
• $\text{SharedPorts}(F', G) \subseteq \text{SharedPorts}(F, G)$

(b) Similarly, it can be shown that $H_{(F'\parallel G)} \subseteq H_{(F\parallel G)}$. This is since $H_{F'} \subseteq H_F$ and $\text{SharedPorts}(F', G) \subseteq \text{SharedPorts}(F, G)$.

(c) The third condition requires that $F \parallel G$ and $F' \parallel G$ to be connected to the same interconnect, which obviously they are. This is since $\text{SharedPorts}(F', G) \subseteq \text{SharedPorts}(F, G)$ and we know that for set of $\text{SharedPorts}(F, G)$ there exists a valid interconnect and thus it is valid for $\text{SharedPorts}(F', G)$.

(d) We should also show that for each dependency set $\nu \in U_{(F\parallel G)}$, either all of its dependencies are independent of $P_{(F\parallel G)}$ or there exists a dependency rule $\mu \in U_{(F\parallel G)}$ such that all of dependencies in $\nu$, defined on $P_{(F\parallel G)}$, also exist in $\mu$. Note that since $F' \preceq F$, for any dependency set $a \in U_{F'}$, either $a$ is independent of $P_F$ or have a corresponding dependency set $b \in U_F$, such that $a$ does not have more dependencies on $P_F$ than $b$ does. In this way, any $\nu$ would comprise of such $a$'s combined with one of $G$'s dependency sets. Observe that the combination of such dependency sets can not introduce extra dependencies on $P_{(F\parallel G)}$. This can be justified by considering the fact that new dependencies can only be introduced through shared ports and we know that $\text{SharedPorts}(F', G) \subseteq \text{SharedPorts}(F, G)$. Thus, $F' \parallel G$ can not introduce more dependencies on $P_{(F\parallel G)}$ than $F' \parallel G$ itself does.

Having shown that $F' \parallel G$ complies with all refinement criteria, we conclude $(F' \parallel G) \preceq (F \parallel G)$.

2. To prove the second condition of well-formedness for MPDs, we should prove that if $F' \preceq F$ and $F\theta$ is defined then $F'\theta$ is also defined and $F'\theta \preceq F\theta$. 
To show that $F'\theta$ is defined, we should show that all conditions in Section 2.2.1 are respected.

- $(I_\theta \cap P^+_F) \subseteq O_{F'}$: This is obvious since, $(I_\theta \cap P^+_F) \subseteq O_F$ and also $O_F \subseteq O_{F'}$.
- $O_\theta \cap P^{\oplus}_{F'} = \emptyset$: This immediately follows from below facts:
  
  $$
  O_\theta \cap P^{\oplus}_F = \emptyset
  $$
  
  $$
  P^{\oplus}_{F'} \subseteq P^{\oplus}_F
  $$

- Also, $\theta$ does not have any two channels with same targets since if it has such channels then $F\theta$ can not be defined too.

Having shown that $F'\theta$ is defined, we should now show that $F'\theta \preceq F\theta$. $I_{F'\theta} \subseteq I_{F\theta}$ since $I_{F'} \subseteq I_F$; also $O_{F'\theta} \supseteq O_{F\theta}$ since $O_{F'} \supseteq O_F$ and $\theta$ contributes to both $O_{F\theta}$ and $O_{F'\theta}$ equally. Similarly, we have $O^+_{F'\theta} \subseteq O^+_{F\theta}$ since $O^+_{F'} \subseteq O^+_F$. Furthermore, $H_{F'\theta} \subseteq H_{F\theta}$ because $H_{F'} \subseteq H_F$ and $\theta$ contributes equally to $H_{F\theta}$ and $H_{F'\theta}$.

Also, notice that dependency pairs introduced on $F'\theta$ dependency sets, through transitivity, are always subset of their counter parts in $F\theta$. This is because firstly $F' \preceq F$ and secondly $\theta$, on $F'\theta$, does not introduce any new dependencies on $P_F$, which was not introduced on $F\theta$. Nevertheless, we conclude $F'\theta \preceq F\theta$, which concludes our proof for MPDs being interface.

\[\square\]

**Theorem 2.5** For all EPDs $F$ and $G$, either they are not composable or $F \parallel G$ is defined and is equal to $G \parallel F$.

\[\square\]

**Proof Sketch.** Observe that composability definition (Definition 2.10) and composition (Definition 2.14) are symmetric with respect to the two involved EPDs and thus composition is a commutative operation.
Theorem 2.6 Enhanced Stateless Port Dependency Interface is a weak interface model.

\[ \square \]

Proof Sketch. To prove that EPD is a weak interface model, it is sufficient to show that it complies with the well-formedness criteria for interfaces, mentioned in 2.1.1; except for associativity in composition. We should first show that composition is actually commutative, which we have already shown, and then should show how EPDs comply with two well-formedness criteria for interfaces.

1. To prove the first condition of well-formedness for interfaces, we should show that given two composable EPDs: \( F = \langle I_F, O_F, O_F^+, H_F, V_F \rangle \) and \( G = \langle I_G, O_G, O_G^+, H_G, V_G \rangle \), where \( V_F = \{v_{F, 1}^1, v_{F, 2}^2, \ldots, v_{F, n}^n\} \) and \( V_G = \{v_{G, 1}^1, v_{G, 2}^2, \ldots, v_{G, m}^m\} \), and EPD \( F' = \langle I_{F'}, O_{F'}, O_{F'}^+, H_{F'}, V_{F'} \rangle \) where \( V_{F'} = \{v_{F', 1}^1, v_{F', 2}^2, \ldots, v_{F', p}^p\} \), if \( F \parallel G \) is defined and \( F' \preceq F \) then \( F' \parallel G \) is also defined and \( F' \parallel G \preceq F \parallel G \).

To show that \( F' \parallel G \) is defined, we should show that \( F' \) and \( G \) comply with the three composability conditions mentioned in Definition 2.10.

(a) \( F' \) and \( G \) are connected to the same interconnect since \( F \) and \( G \) are connected to the same interconnect, because they are composable and, furthermore, \( F' \) refines \( F \) and thus is connected to the same interconnect as \( F \).

Also, composability between two EPDs requires that each port belonging to the set of shared ports should belong to input port of one and output of the other EDP. Furthermore, there should exist a channel in two EPDs’ mutual interconnect such that it creates each of those shared ports. Notice that \( \text{SharedPorts}(F', G) \) does not introduce more shared ports than \( \text{SharedPorts}(F, G) \) does. This is since \( F' \) can only have less input ports and more output ports, all such ports belong to \( O_{F}^+ \) though.\(^3\) In this way, obviously \( \text{SharedPorts}(F', G) \) can not have a bigger set of shared ports than \( \text{SharedPorts}(F, G) \). In effect, since this condition holds for \( F \) and \( G \) it holds for \( F' \) and \( G \) too.

\(^3\)This follows from refinement definition in stateless input/output interfaces where it states \( O_{F'}^+ \subseteq O_{F}^+ \), see Section 2.1.2 for more details.
(b) Secondly, we should show that $H_F \cap H_G = \emptyset$, which follows from the following two facts: $H_F \cap H_G = \emptyset$, by compositability of $F$ and $G$, and $H_{F'} \subseteq H_F$, by refinement definition.

(c) Thirdly, we should show that non of different combinations of $F'$ and $G$ dependency rules result cycles of dependencies. To show that, observe that a cycle of such dependencies, $\langle (a_0, b_0), (a_1, b_1), ..., (a_n, b_n) \rangle$, can exist only if $b_n = a_0$ and $b_j = a_{j+1}$ for all $0 \leq j < n$. This implies that all ports involved in such a sequence would belong to $\text{SharedPorts}(F', G)$. On the other hand $\text{SharedPorts}(F', G) \subseteq \text{SharedPorts}(F, G)$, and $F'$ can not introduce more dependencies than $F$ have introduced on such ports (by refinement definition). Thus, such a cycle of dependencies could only exist for $F'$ and $G$ only if it exists for $F$ and $G$ too. But this is not true since $F$ and $G$ are composable. Nevertheless, such a cycle of dependencies does not exist for $F'$ and $G$ either.

Having shown that $F'$ and $G$ meet the compositability requirements, we conclude $F' \parallel G$ is defined.

Now we should demonstrate that $F' \parallel G \preceq F \parallel G$. We show how $F' \parallel G$ complies with three requirements, mentioned in Section 2.3.3, for being refinement of $F \parallel G$.

(a) For the first refinement condition, we should show $F' \parallel G$ does not have more input ports or less output ports that $F \parallel G$ does.

As for the input ports, we know that since $F'$ refines $F$, then $(I_{F'} \cup I_G) \subseteq (I_F \cup I_G)$. On the other hand, notice that $\text{SharedPorts}(F', G) \subseteq \text{SharedPorts}(F, G)$ and also observe that $(I_{F'} \cup I_G) \cap (\text{SharedPorts}(F, G) \setminus \text{SharedPorts}(F', G))$ is an empty set.\footnote{Assume this set is not empty and $a \in ((I_{F'} \cup I_G) \cap (\text{SharedPorts}(F, G) \setminus \text{SharedPorts}(F', G)))$. Then, since $a \in \text{SharedPorts}(F, G)$ it should be the case that $a \in \text{SharedPorts}(F', G)$ too; this is because we know that $a \in (I_{F'} \cup I_G)$ and also we know that $(O_F \cup O_G) \subseteq (O_{F'} \cup O_G)$. But we know that $a \notin \text{SharedPorts}(F', G)$ and thus such an $a$ can not exist.} These two facts imply that although $\text{SharedPorts}(F', G)$ is a subset of $\text{SharedPorts}(F, G)$ but extra members in $\text{SharedPorts}(F, G)$ do not
have any counter parts in $I_F \cup I_G$ and thus we can conclude:

$$(I_F \cup I_G) \setminus \text{SharedPorts}(F', G) \subseteq (I_F \cup I_G) \setminus \text{SharedPorts}(F, G)$$

On the other hand, $F' \preceq F$ implies that $O_F^+ \supseteq O_F$ and it follows that:

$$(O_F \cup O_G) \setminus \text{SharedPorts}(F', G) \supseteq (O_F \cup O_G) \setminus \text{SharedPorts}(F, G)$$

Finally, since $O_F^+ \subseteq O_F^+$ we can conclude:

$$(O_F^+ \cup O_G^+) \setminus \text{SharedPorts}(F', G) \subseteq (O_F^+ \cup O_G^+) \setminus \text{SharedPorts}(F, G)$$

(b) Secondly, we should prove that $H_{F'\parallel G} \subseteq H_{F\parallel G}$, which is obvious from the following facts:

- $H_{F\parallel G} = H_F \cup H_G \cup \text{SharedPorts}(F, G)$
- $H_{F'\parallel G} = H_{F'} \cup H_G \cup \text{SharedPorts}(F', G)$
- $\text{SharedPorts}(F', G) \subseteq \text{SharedPorts}(F, G)$
- $H_{F'} \subseteq H_F$

(c) Thirdly, we should show that $V_{F'\parallel G} \subseteq V_{F\parallel G}$. But before doing that, we should show that $F' \parallel G$ and $F \parallel G$ are in fact connected to the same connection. By composability definition, $F$ and $G$ are connected to the same interconnect and similarly $F'$ and $G$ are connected to the same interconnect and thus $F' \parallel G$ and $F \parallel G$ are connected to the same interconnect.

We should show that $V_{F'\parallel G} \subseteq V_{F\parallel G}$. In order to show that, by Definition 2.7, we should show that any $\nu \in V_{F'\parallel G}$ either does not have any common dependencies with any of dependencies in $V_{F\parallel G}$ or if it has, there exists a $\mu \in V_{F\parallel G}$, such that: firstly, $\nu$ does not have more dependencies on $P_{F\parallel G} \times P_{F\parallel G}$ than $\mu$ has and secondly the sequence of $\nu$ dependencies on $P_{F\parallel G} \times P_{F\parallel G}$ does not violate $\mu$'s pattern.

To show that all dependencies in $V_{F'\parallel G}$ adhere to above conditions, suppose $\nu \in V_{(F',G)}$. If $\nu$ does not have any dependencies on $P_{F\parallel G} \times P_{F\parallel G}$ then $\nu$ is a valid dependency. If it has such dependencies then suppose $\nu$ is the result of
two dependency rules $v^i_F$ and $v^j_G$. Following, we provide different possibilities
for $v^i_F$ and $v^j_G$ and their having dependencies on $P_{F\parallel G} \times P_{F\parallel G}$ and would show
that for all such $\nu$ s, there are some dependency rules which $\nu$ respects their
dependencies. Observe that other possibilities are not valid since they will not
create dependency rules with some dependencies on $P_{F\parallel G}$.

1. \[((\text{set}(\text{Domain}(v^i_F)) \cup \text{set}(\text{Range}(v^i_F))) \cap P_{F\parallel G} = \emptyset) \land
((\text{set}(\text{Domain}(v^j_G)) \cup \text{set}(\text{Range}(v^j_G))) \cap P_{F\parallel G} \neq \emptyset)\]

This possibility itself leads to two different possibilities based on whether
$v^i_F$ and $v^j_G$ have dependencies on $\text{SharedPorts}(F, G)$ or not. Notice that
any other possibilities would fail to create a valid dependency rule for $F' \parallel G$
because they will have dependencies on shared ports which is not valid.

- \[((\text{set}(\text{Domain}(v^i_F)) \cup \text{set}(\text{Range}(v^i_F))) \cap \text{SharedPorts}(F, G) = \emptyset) \land
((\text{set}(\text{Domain}(v^j_G)) \cup \text{set}(\text{Range}(v^j_G))) \cap \text{SharedPorts}(F, G) = \emptyset)\]

Since $v^i_F$, neither has any shared dependency rules on $P_{F\parallel G}$ nor on
$\text{SharedPorts}(F, G)$, $\nu = v^{(i,j)}_{(F',G)}$ can create its dependencies on $P_{F'\parallel G}$
through $v^j_G$ only. In fact, such $\nu$ starts with all dependencies of $v^i_F$, followed
by all dependencies in $v^j_G$. We claim for such a $\nu \in \mathcal{V}_{F'\parallel G}$, there
exists at least one $\mu$ such that $\nu$ respects it. Consider $\mu = v^{k,j}_{(F,G)} \in
\mathcal{V}_{F\parallel G}$, where $1 \leq k \leq n$, and $v^k_F$ does not have any dependencies on
$\text{SharedPorts}(F, G)$. If such $v^k_F$ exists, we claim $\nu$ respects such $\mu$ s’
dependencies. This is since $v^k_F$ does not have any dependencies on shared
ports and thus $\mu$ could be created by using dependencies of $v^k_F$ followed
by $v^j_G$. Such a $\mu$ can be obviously be respected by $\nu$ which does not
have any extra dependencies on $P_{F\parallel G}$ than $\mu$ has and furthermore does
not violate the sequence of dependencies that $\mu$ has. However, it could
be the case that such a $v^k_F$ does not exist, then we claim that $v^j_G$ is a
closed dependency rule with respect to $F$ and there exists $\mu = v^{j,j}_{(G,G)}$
which can be obviously respected by $\nu$.

- \[(\text{set}(\text{Domain}(v^i_F)) \cup \text{set}(\text{Range}(v^i_F))) \cap \text{SharedPorts}(F, G) \neq \emptyset \land
(\text{set}(\text{Domain}(v^j_G)) \cup \text{set}(\text{Range}(v^j_G))) \cap \text{SharedPorts}(F, G) \neq \emptyset)\]
Since \( F' \preceq F \) and \( v^i_{F'} \) has some dependencies on \( \text{SharedPorts}(F,G) \), there should exist a \( v^k_F \in \mathcal{V}_F \), where \( 0 \leq k \leq n \), such that \( v^i_{F'} \) respects \( v^k_F \) dependencies. Now, consider \( \mu = v^{k,j}_{(F,G)} \in \mathcal{V}_{F||G} \), we claim that \( \nu \) respect such a \( \mu \)'s dependencies. This is since \( v^i_{F'} \) respects \( v^k_F \) and thus, in conjunction with \( v^j_G \), it can not create more dependencies than \( v^k_F \) does. Furthermore, the sequence of created dependencies complies with such a \( \mu \). Also, note that if such a \( \mu \) is not defined due to existence of shared ports in the resulting dependency rule, then \( \nu \) can not be defined for the same reason either.

\[ \text{ii. } (\text{set}(\text{Domain}(v^i_{F'}))) \cup \text{set}(\text{Range}(v^i_{F'}))) \cap \mathcal{P}_{F||G} \neq \emptyset \]

Again, this possibility suggests two classes of possibilities. Depending on whether \( v^i_{F'} \) and \( v^j_G \) have dependencies on \( \text{SharedPorts}(F,G) \) or not, we show how there exists some dependency rule such that \( \nu \) respects them. Other possibilities are not allowed since they fail creating valid dependency rules for \( F' \parallel G \), due to having dependencies on shared ports.

- \( (\text{set}(\text{Domain}(v^i_{F'}))) \cup \text{set}(\text{Range}(v^i_{F'}))) \cap \mathcal{P}_{F||G} = \emptyset \land (\text{set}(\text{Domain}(v^j_G))) \cup \text{set}(\text{Range}(v^j_G))) \cap \mathcal{P}_{F||G} = \emptyset \)

This suggests that the two dependency rules \( v^i_{F'} \) and \( v^j_G \) do not have any shared ports involved in their dependency rules and thus \( \nu \) simply consists of \( v^i_{F'} \)'s dependencies followed by \( v^j_G \)'s. On the other hand, since \( F' \preceq F \) and \( v^i_{F'} \) has some dependencies on \( \mathcal{P}_{F||G} \), there should exist a \( v^k_F \in \mathcal{V}_F \), where \( 0 \leq k \leq n \), such that \( v^i_{F'} \) respects \( v^k_F \) dependencies. We claim that \( \nu \) respects dependencies belonging to \( \mu = v^{k,j}_{(F,G)} \in \mathcal{V}_{F||G} \).

This is since \( \mu \) is comprised of \( v^k_F \)'s dependencies followed by \( v^j_G \)'s and thus \( \nu \) obviously respects its dependencies.

- \( (\text{set}(\text{Domain}(v^i_{F'}))) \cup \text{set}(\text{Range}(v^i_{F'}))) \cap \mathcal{P}_{F||G} \neq \emptyset \land (\text{set}(\text{Domain}(v^j_G))) \cup \text{set}(\text{Range}(v^j_G))) \cap \mathcal{P}_{F||G} \neq \emptyset \)

Since \( F' \preceq F \) and \( v^i_{F'} \), has dependencies on \( \mathcal{P}_{F||G} \), we concluded there should exist a \( v^k_F \in \mathcal{V}_F \), where \( 0 \leq k \leq n \), such that \( v^i_{F'} \) respects \( \mathcal{V}^k_F \) dependencies. Notice that \( \nu \)'s dependencies, incurred by combin-
ing dependencies in $v^i_{F'}$ and $v^j_G$, can not introduce more dependencies on $P_{F\parallel G}$ than $\mu = v^{k,i}_{(F,G)}$ does. This is since dependencies involving shared ports on $v^i_{F'}$ are all present in $v^k_{F}$. However, there could exist some incurred dependencies in $\nu$ which do not exist in $\mu$; but those dependencies are not on $P_{F\parallel G}$ and are essentially generated since $\text{SharedPorts}(F', G) \subseteq \text{SharedPorts}(F, G)$. Also, notice that if $\mu$ is not defined, due to existence of shared ports in some dependencies, then $\nu$ can not be defined either. Nevertheless $\nu$ obviously respects $\mu$’s dependencies.

Above possibilities essentially represent all different possibilities by which $\nu$ can be created. We showed that in all conditions, $\nu$’s sequence of dependencies respect some dependency rule $\mu \in \mathcal{V}_{F\parallel G}$.

Having shown that regardless of the type of dependency rules in $\mathcal{V}_{F'}$, all dependency rules incurred by $\mathcal{V}_{(F', G)}$ respect the dependencies in $\mathcal{V}_{F\parallel G}$, we should now show that any $\eta \in \mathcal{V}_{(G, F')}$ also respect $\mathcal{V}_{F\parallel G}$ dependencies. However, that argument is quite symmetric with the argument we provided for dependencies in $\mathcal{V}_{(F', G)}$ and thus we avoid the redundancy. Nevertheless, all dependency rules in $\mathcal{V}_{F'\parallel G}$ respect some dependencies in $\mathcal{V}_{F\parallel G}$ and thus $\mathcal{V}_{F'\parallel G} \subseteq \mathcal{V}_{F\parallel G}$.

Satisfying all necessary conditions for refinement, we conclude that $F' \parallel G \preceq F \parallel G$.

2. To prove the second condition of well-formedness for interfaces, we should prove that if $F' \preceq F$ and $F\theta$ is defined then $F'\theta$ is also defined and $F'\theta \preceq F\theta$.

To show that $F'\theta$ is defined, it should comply with EPD connection conditions in Section 2.3.2, page 34. Following, we show how $F'\theta$ complies with those conditions.

- $(I_\theta \cap P^+_F) \subseteq O_{F'}$
  This is obvious since, $(I_\theta \cap P^+_F) \subseteq O_F$ and also $O_F \subseteq O_{F'}$.

- $O_\theta \cap P^+_F = \emptyset$
  This immediately follows from below facts:
  
  $O_\theta \cap P^\oplus_F = \emptyset$
  $P^\oplus_{F'} \subseteq P^\oplus_F$
• Also, $\theta$ does not have any two channels with the same target since if it has such channels then $F\theta$ cannot be defined too.

Having shown that $F'\theta$ is defined, we can now prove $F'\theta \preceq F\theta$. From $F' \preceq F$, the following facts follow:

\[
\begin{align*}
I_{F'} & \subseteq I_F \\
O^+_{F'} & \subseteq O^+_F \\
O_{F'} & \supseteq O_F \\
H_{F'} & \subseteq H_F \\
V_{F'} & \subseteq V_F
\end{align*}
\] (A.5)

On the other hand, by definition of connection (see Section 2.3.2), the input set of an EPD enhanced by a connection remains the same while its output set can be increased by the output set of that connection. Thus from Equation A.5, we have the following:

\[
\begin{align*}
I_{F'\theta} & \subseteq I_{F\theta} \\
O^+_{F'\theta} & \subseteq O^+_{F\theta} \\
O_{F'\theta} & \supseteq O_{F\theta}
\end{align*}
\] (A.6)

Furthermore, interconnect $\theta$ can not introduce more reserved ports on $F'$ than it does on $F$. Thus,

\[
H_{F'\theta} \subseteq H_{F\theta}
\] (A.7)

On the other hand, considering any dependency rule $v^i_F \in V_{F'}$, where $0 \leq i \leq p$, and the dependency rule $v^k_F \in V_F$, where $0 \leq k \leq n$, such that $V_{F'}$ respects $V_F$, interconnect $\theta$ does not change such corresponding respect relations. This is since any additional incurred dependency on $V_{F'}$ would either also appear in $V_F$, in the appropriate order, or would be on sets of ports irrelevant to $V_F$ which again does not hurt the respect relation. Thus,

\[
V_{F'\theta} \subseteq V_{F\theta}
\] (A.8)

Equations A.6, A.7 and A.8 amount to the refinement definition of $F'\theta \preceq F\theta$ and this concludes this part of proof.
By proving the two properties of interfaces for EPD and also showing that composition is commutative in EPD, we can conclude that EPD is a weak interface and this concludes our proof.

□

A.2 Chapter 3 Proofs

Lemma 3.4  For each state \((v, u) \in V_{P\#Q}\), if \((v, u)\) is a complex state then there is not more than one step in \(\tau_{P\#Q}\) such that it initiates from \((v, u)\).

□

Proof. Let \((v, u) \in V_{P\#Q}\) be such a state with at least one step initiating from it. We show that there could exactly exist one such step. We present our proof in three stages based on whether \(v\), \(u\) or both are complex states:

1. \(v \in V_P^{\text{comp}}\) and \(u \in V_Q^{\text{abst}}\): According to Definition 3.10, only sets (3) and (4) can contribute to creating steps that initiate from such states. Note that both sets require that if there exists a step \(((v, u), a, (v', u')) \in \tau_{P\#Q}\) there should also exist \((v, a, v') \in \tau_P\). On the other hand, \(v\) is a complex state which implies that there is only one such step in \(\tau_P\). Nevertheless, we can conclude that in this case there can not exist more than one step initiating from \((v, u)\).

2. \(v \in V_P^{\text{abst}}\) and \(u \in V_Q^{\text{comp}}\): Based on Definition 3.10, only sets (3) and (5) can add steps which initiate from such \((v, u)\). Quite similar to the observation in the previous case, we can conclude it is not possible to have more than one step initiate from \((v, u)\).

3. \(v \in V_P^{\text{comp}}\) and \(u \in V_Q^{\text{comp}}\): In this case only set (3) in Definition 3.10 can create steps such that they initiate from \((v, u)\) which obviously implies that there could not exist more than one step initiating from \((v, u)\). This follows from the fact that \(P\) and \(Q\) do not have more than one step initiating from \(u\) and \(v\) respectively.

□
**Lemma 3.5** For each state \((v, u) \in V_{P*Q}\), if \((v, u)\) is a complex state then, there is not more than one step in \(\tau_{P*Q}\) such that it ends in \((v, u)\).

**Proof.** Assume state \((v, u)\) such that there exists at least one step terminating in it. We show that there exists exactly one step terminating at that state. Depending on whether \(v\) or \(u\) or both are complex states, we present our proof in three stages:

1. \(v \in V_{P}^{\text{comp}}\) and \(u \in V_{Q}^{\text{abst}}\): According to Definition 3.10, only sets identified as (1), (3) and (4) can contribute in creating steps which terminate in \((v, u)\). Furthermore, in this case, it can be observed that only one of the three aforementioned sets can contribute to adding steps to \(\tau_{P*Q}\). Also, any steps \(((v', u'), a, (v, u))\) can exist in \(\tau_{P*Q}\) only if \((v', a, v) \in \tau_{P}\), which by definition, is the only step terminating in \(v\). Nevertheless, in case of sets (1) and (4), since we have \(u = u'\) we conclude that those sets can not contribute more than one step to \(\tau_{P*Q}\). In case of set (3), again only one step can be added to \(\tau_{P*Q}\) which terminates in \((v, u)\). This is since, by definition, both \((v', a, v) \in \tau_{P}\) and \((u', a, u) \in \tau_{Q}\) are complex steps and thus do not introduce more than one step terminating in \(v\) and \(u\) respectively.

2. \(v \in V_{P}^{\text{abst}}\) and \(u \in V_{Q}^{\text{comp}}\): In this case, only sets identified as (2), (3) and (5) can contribute in creation of \(\tau_{P*Q}\). Furthermore, the same mutual exclusion relation mentioned in the previous section can be observed among these sets. The proof for this case is symmetric with the one presented for the case when \(v\) was a complex state and the reader is referred to that section.

3. \(v \in V_{P}^{\text{comp}}\) and \(u \in V_{Q}^{\text{comp}}\): Such a \((v, u)\) can only create steps that are introduced by set (3) in Definition 3.10. Thus, it can only participate in steps with shared actions. Assume \(((v', u'), a, (v, u))\) as a step in \(\tau_{P*Q}\), both \((v', a, v) \in \tau_{P}\) and \((u', a, u) \in \tau_{Q}\) are complex steps and thus do not introduce more than one step terminating in \(v\) and \(u\), respectively. In effect, similar to the other cases, there could not exist more than one step terminating in \((v, u)\) and this concludes our proof.

\(\square\)
Lemma 3.7  For each complex state \((v, u) \in V_{P \otimes Q}\) there exists exactly one sequence \(\langle (v_0, u_0), a_0, (v_1, u_1), \ldots, (v_n, u_n) \rangle\), represented as \(\Delta(v, u)\), such that the following conditions are true:

- \(\exists (v_j, u_j) \cdot ((0 < j < n)) \land (v_j = v) \land (u_j = u)\)
- All \(\langle (v_i, u_i), a_i, (v_{i+1}, u_{i+1}) \rangle \in \tau_{P \otimes Q}\), where \(0 \leq i < n\), are complex steps
- \((v_0, v_n \in V_{P}^{\text{abst}}) \land (u_0, u_n \in V_{Q}^{\text{abst}})\)

\(\Box\)

Proof. From Lemmas A.2 and A.2, it follows that each complex state \((v, u) \in V_{P \otimes Q}\) can have at most one step initiating and terminating at it. Furthermore, since any such state, according to \(V_{P \otimes Q}\) definition, have exactly one step initiating from it, we conclude that any complex fragment would eventually terminate in a non-complex state, i.e. a state \((v, u)\) where \(v \in V_{P}^{\text{abst}}\) and \(u \in V_{Q}^{\text{abst}}\).

On the other hand, let’s assume that there does not exist any \((v_0, u_0)\) as described above. Then, we argue that for any reachable complex state \((v, u) \in V_{P \otimes Q}\), there exists an initial state \((x, y) \subseteq V_{P}^{\text{init}} \times V_{Q}^{\text{init}}\), as a non-complex state, such that \((v, u)\) can be reached from that state which can be assumed as \((v_0, u_0)\). Nevertheless, we conclude that there exists a \(\langle (v_0, u_0), a_0, (v_1, u_1), \ldots, (v_n, u_n) \rangle\) sequence such that the above conditions hold.

\(\Box\)

Theorem 3.8  Given a complex state \((v, u) \in V_{P \otimes Q}\) and \(s = \Delta(v, u)\), an alternating sequence of states and actions, only one of the followings is true:

- \(\exists x \in \tau_{P}^{\text{abst}} \cdot \text{sched}(\phi_P(x)) = \text{sched}(s)\)
- \(\exists y \in \tau_{Q}^{\text{abst}} \cdot \text{sched}(\phi_Q(y)) = \text{sched}(s)\)
- \(\exists x \in \tau_{P}^{\text{abst}}, \exists y \in \tau_{Q}^{\text{abst}} \cdot (\text{sched}(\phi_P(x)) = \text{sched}(s)) \land (\text{sched}(\phi_Q(y)) = \text{sched}(s)) \land (\text{sched}(\phi_P(x)) = \text{sched}(\phi_Q(y)))\)
Proof. Assume $s = \Delta(v, u)$ to be an alternating sequence $\langle (v_0, u_0), a_0, (v_1, u_1), \ldots, (v_n, u_n) \rangle$. According to $\Delta$ definition, for all $(v_i, u_i)$, where $(0 < i < n)$, either $v_i \in V^\text{comp}_P$ or $u_i \in V^\text{comp}_Q$. Now consider the first complex state $(v_j, u_j)$, such that only one of $v_j$ or $u_j$ is a complex state, if such a state exists at all. Depending on whether $v_j$ or $u_j$ is a complex state, the sequence $s$ at $(v_j, u_j)$ will be following the schedule of the abstract step $\text{step}(v_j) \in \tau^\text{abst}_P$ or $\text{step}(u_j) \in \tau^\text{abst}_Q$, at $v_j$ or $u_j$ respectively. This can be observed by inspecting $\tau^\text{P}\#Q$ definition in Definition 3.10; notice that any steps initiating from such a $(v_j, u_j)$ would be created by either set (4) or (5) in Definition 3.10. Furthermore, the next state, i.e. $(v_{j+1}, u_{j+1})$, where $(j + 1) < n$, depending on whether $v_j$ or $u_j$ is a complex state, should certainly have either $v_{j+1}$ or $u_{j+1}$ as a complex state respectively. It can be observed that for all $(v_k, u_k)$, where $j < k < n$, if $v_j$ is a complex state then all $v_k$s are complex states. Conversely, if $u_j$ is a complex state then all $u_k$s are complex states. In this way, according to $\tau^\text{P}\#Q$ definition, all $(v_k, u_k)$s should initiate a step with the same action as the step which $v_k$ or $u_k$ initiates. It is also possible to have a shared action enabled at $(v_k, u_k)$ which means both $v_k$ and $u_k$ initialize steps with the same actions. However, this does not affect the observation that we made earlier about $v_k$s and $u_k$s, and their being complex states based on whether $v_j$ or $u_j$ is a complex state. We also notice that $(v_k, u_k)$ could have both $v_k$ and $u_k$ as complex states. However, such states can only initiate steps with shared actions and thus again respect the observations that all $(v_k, u_k)$ initiate steps with similar actions as $v_k$ or $u_k$ initiate.

We demonstrated that all $(v_k, u_k)$ states comply with a schedule of an abstract step in $P$ or $Q$. For state $(v_n, u_n)$, observe that both $v_n$ and $u_n$ should be non-complex states. This is true since by composability conditions two composable PIAs can not have abstract steps with overlapping complex fragments. This condition avoids the situation where a complex fragment from $P$ ends and another interleaved complex fragment from $Q$ has more steps to finish, or vice versa.

Now, we should show that all $(v_l, u_l)$ states, where $(0 \leq l < j)$, also comply the execution schedules that $(v_k, u_k)$s follow. Consider $(v_{j-1}, u_{j-1})$ where $(j - 1) \geq 0$, and let’s assume that $v_j$ is a complex state, then according to $\tau^\text{P}\#Q$ definition, $(v_{j-1}, a_{j-1}, v_j) \in \tau_P$ and $((v_{j-1}, u_{j-1}), a_{j-1}, (v_j, u_j)) \in \tau^\text{P}\#Q$. Conversely, if we assume that $u_j$ is a complex step
then it follows that \((u_{j-1}, a_{j-1}, u_j) \in \tau_Q\) and \(((v_{j-1}, u_{j-1}), a_{j-1}, (v_j, u_j)) \in \tau_{P \# Q}\). In other words, \((v_{j-1}, u_{j-1})\) execution schedule rely on \((v_j, u_j)\) and whether \(v_j\) or \(u_j\) is a complex state. Similarly, we can show for all other \((v_l, u_l)\)s, that they rely on \((v_j, u_j)\) for their execution schedule.

Considering state \((v_1, u_1)\), based on \(v_j\) or \(u_j\) being complex states, either \(v_1\) or \(u_1\) is a complex state. Then, according to \(\tau_{P \# Q}\) definition, there should exist either \((v_0, a_0, v_1)\) or \((u_0, a_0, u_1)\). In either cases, since we know that \(v_0\) and \(u_0\) are non-complex steps, we can conclude that such a step is the first step in a complex fragment in \(P\) or \(Q\); because either \(v_1\) or \(u_1\) is a complex state. Nevertheless, we can conclude that \(sched(s)\) is equal to a schedule of a complex fragment in \(P\) or \(Q\).

We notice that, there may not exist a \((v_j, u_j)\) as described above. That would mean that all \((v_m, u_m)\)s, where \(0 < m < n\), are complex states. According to \(\tau_{P \# Q}\) definition, however, only set (3) can initiate steps from such \((v_m, u_m)\)s, which implies that two abstract steps from \(P\) and \(Q\) completely overlap each other and this is an acceptable case stated as the last case of this theorem. \(\square\)

**Theorem 3.11** Given two PIAs \(P\) and \(Q\), their composability and their composition \(P \parallel Q\), can be computed in time linear in \(|P| \times |Q|\). \(\square\)

**Proof.** According to observations 3.1 and 3.2, respectively in pages 71 and 74, the composability of two PIAs can be carried out in maximum time linear in \(|P| \times |Q|\). Step 1 of the above algorithm can be carried out in \(|P| \times |Q|\) too, see observation 3.3 in page 75. Illegal states of \(V_{P \# Q}\), step 2, can also be computed in linear time as shown in [dAH01a]. On the other hand, algorithm 3.11 computes \(V_{P \# Q}\) and \(\tau_{P \# Q}\) in time linear in \(|P| \times |Q|\), see Observation 3.6 in page 76 for justifications. Finally, \(\Psi_{P \# Q}\) can be computed in time linear in \(|P| \times |Q|\), see observations 3.9 and 3.10 in pages 78 and 78 respectively.

Also, as mentioned earlier, it is always possible to remove unreachable states and steps in a PIA in linear time in its size. This means step 7 in computation of \(P \parallel Q\) does not add more than \(|P| \times |Q|\) complexity to the whole process.

Nevertheless, the whole computation consists of steps that can be carried out, at most, in time linear in \(|P| \times |Q|\) and thus \(P \parallel Q\) can be carried out in linear time in \(|P| \times |Q|\).
Theorem 3.12  Given two composable PIAs $P$ and $Q$, $P \parallel Q = Q \parallel P$. □

Proof Sketch. Considering the steps required to be carried out for computing composition of two PIAs, it can be observed that all of those steps are comprised of symmetric computations. Furthermore, the involved operations in such computations are all commutative.

The above argument can be validated by inspecting: definition of interleaved steps and states in Definition 3.10, definition of illegal states in Figure 3.4, definition and algorithm for computation of legal interleaved steps and states in Definition 3.12 and Figure 3.11, definition of first states and steps of the complex steps in Definition 3.13, definition of mapping function $\Psi$ on the set of first states of the complex states as defined in Definition 3.14 and finally computation of the product of two PIAs as defined in Section 3.3.2. □

Theorem 3.13  Given three PIAs $P$, $Q$ and $R$, either some of PIAs are not composable or $(P \parallel Q) \parallel R = P \parallel (Q \parallel R)$. □

Proof Sketch. If none of the involved PIAs in a composition has complex actions, then it is easy to show that PIA composition is equivalent to IA composition. Since we know that IA composition is an associative operation then it would suffice to show that PIA composition remains associative in presence of complex actions too. Below, we only provide the outline of such a proof.

To prove that associativity holds, we should first show that the sets of interleaved states and steps generated through $(P \parallel Q) \parallel R$ and $P \parallel (Q \parallel R)$ actually result the same sets. This fact can be observed by inspecting Definition 3.10 and also realizing that $\text{SimpShared}(P,Q)$, $\text{SimpShared}(P,R)$ and $\text{SimpShared}(Q,R)$ are disjoint sets. This can be observed by noting that the three PIAs are composable. Having disjoint sets of simple shared actions, Definition 3.10 creates similar sets of interleaved states and steps, regardless of the order of necessary compositions to be carried out.

Secondly, we should show that the sets of illegal states and illegal complex states would
amount to the same value regardless of the order of compositions. In fact, it should be shown that eventually all illegal and illegal complex states will be removed no matter in what order we do the compositions. Again, this can be observed by noticing that simple and complex shared actions are disjoint among three PIAs and thus at some stage of composition illegal states, both complex and simple, would show up.

Thirdly, it should be shown that the computation of $\Psi$ is independent of the order of involved PIAs’ composition. Computation of $\Psi$ is obviously independent of order of composition; a tricky situation is when there are some mutual embedding actions among $P$, $Q$ and $R$. As an example of such a situation, suppose there is $a \in \text{Embedded}(P, Q)$ and $a \in A^\text{comp}_Q$, and also there exists $b \in A^\text{comp}_P$ such that $\phi_Q(a) \subseteq \phi_P(b)$. Also, suppose $c \in \text{Embedded}(P, R)$ such that $\phi_R(c) \subseteq \phi_P(b)$. Observe that $b$ is a mutual embedding action, however, even in this situation the resulting complex action incurred by composition is always unambiguously mapped to $b$; for more on $\Psi$ computation, see Definition 3.14.

Nevertheless, it can be inferred that composition in PIA is an associative operation. □

Theorem 3.14 Given three PIAs $P$, $Q$ and $P'$ such that $P' \succeq P$, $P$ and $Q$ are composable and $P'$ and $Q$ are composable too, then it should be true that $(P \parallel Q) \succeq (P' \parallel Q)$. □

Proof Sketch. To prove this property, there are two main issues to be proved. Firstly, we should show that $(P' \parallel Q)$ has more inputs and less outputs than $(P \parallel Q)$, as required in Definition 3.20. Secondly, we should show that the alternating simulation holds between the initial states of the two compositions.

To show the first part of the proof, we first show that $A^I_{P||Q}$ can not be smaller than $A^I_{P'||Q}$. $A^I_{P||Q}$ can be computed as following:

$$A^I_{P||Q} = A^I_{P} \cup A^I_{Q} \setminus \text{SimpShared}(P, Q)$$

$$= A^I_{P} \cup A^I_{Q} \setminus ((A^I_{P} \cap A^O_{Q}) \cup (A^O_{P} \cap A^I_{Q}))$$

Similarly, $A^I_{P'||Q}$ can be computed as following:

$$A^I_{P'||Q} = A^I_{P'} \cup A^I_{Q} \setminus ((A^I_{P'} \cap A^O_{Q}) \cup (A^O_{P'} \cap A^I_{Q}))$$
It is now possible to compare the constituent elements of $\mathcal{A}_{P'\|Q}^{I,simp}$ and $\mathcal{A}_{P\|Q}^{I,simp}$ against each other. Considering the following facts, that are followed by refinement definition between $P'$ and $P$:

$$(\mathcal{A}_{P'}^{I,simp} \cup \mathcal{A}_{Q}^{I,simp}) \supseteq (\mathcal{A}_P^{I,simp} \cup \mathcal{A}_Q^{I,simp})$$

$$(\mathcal{A}_{P'}^{O,simp} \cap \mathcal{A}_{Q}^{I,simp}) \subseteq (\mathcal{A}_P^{O,simp} \cap \mathcal{A}_Q^{I,simp})$$

It suffices to compare $\mathcal{A}_{P'}^{I,simp} \setminus (\mathcal{A}_{P'}^{I,simp} \cap \mathcal{A}_{Q}^{O,simp})$ against $\mathcal{A}_{P}^{I,simp} \setminus (\mathcal{A}_P^{I,simp} \cap \mathcal{A}_Q^{O,simp})$. This comparison can be reduced to the comparison of $(\mathcal{A}_{P'}^{I,simp} \setminus \mathcal{A}_{Q}^{O,simp})$ against $(\mathcal{A}_P^{I,simp} \setminus \mathcal{A}_Q^{O,simp})$. Such a comparison would result:

$$(\mathcal{A}_{P'}^{I,simp} \setminus \mathcal{A}_{Q}^{O,simp}) \supseteq (\mathcal{A}_P^{I,simp} \setminus \mathcal{A}_Q^{O,simp})$$

which implies:

$$\mathcal{A}_{P'\|Q}^{I,simp} \supseteq \mathcal{A}_{P\|Q}^{I,simp}$$

Similarly, it can be shown that:

$$\mathcal{A}_{P'\|Q}^{O,simp} \subseteq \mathcal{A}_{P\|Q}^{O,simp}$$

It should also be shown that $\lambda_{P\|Q}^{I} \subseteq \lambda_{P'\|Q}^{I}$ and $\lambda_{P\|Q}^{O} \supseteq \lambda_{P'\|Q}^{O}$ which can be proved by considering the following facts:

$$\mathcal{A}_{P'\|Q}^{I,simp} \supseteq \mathcal{A}_{P\|Q}^{I,simp}$$

$$\mathcal{A}_{P'\|Q}^{O,simp} \subseteq \mathcal{A}_{P\|Q}^{O,simp}$$

$$\lambda_{P'}^{I} \supseteq \lambda_{P}^{I}$$

$$\lambda_{P'}^{O} \subseteq \lambda_{P}^{O}$$

In the second part of the proof, it should be shown that there exists an alternating simulation between states of $P' \| Q$ and $P \| Q$, as required in PIA refinement definition. This can be proved by observing the fact that there exists an alternating simulation relation between states of $P'$ and $P$. $P' \| Q$ and $P \| Q$ maintain such a relation since composition maintains the contravariant property of more inputs and less outputs at alternating simulated states. Moreover, it can be shown that there could not exist more simple or complex illegal states in $P' \| Q$ than in $P \| Q$ such that those states violate the alternating simulation relation between the two PIAs. □
Appendix B

Glossary of Symbols

B.1 Chapter 2 Symbols

$P_F$ For a stateless interface $F$, $P_F = I_F \cup O_F$.

$P_F^+$ For a stateless interface $F$, $P_F^+ = I_F \cup O_F^+$.

$P_F^\oplus$ For a stateless interface $F$, $P_F^\oplus = P_F^+ \cup H_F$.

$\text{SharedPorts}(F,G)$ The set of shared ports between two MPDs or EPDs, $F$ and $G$ is equal to $P_F^+ \cap P_G^+$.

$\text{Pos}_s(v)$ Given sequence $s$ and a value $v$, $\text{Pos}_s(v)$ provides the position(s) of that value in the sequence.

$s' \subset s$ $s'$ is subsequence of $s$.

$r \oplus s$ Sequence $s$ is concatenated to the end of $r$. 
$s \ominus t$  
All of the members in sequence $t$ that appear in sequence $s$ are removed from $s$.

$s^\circ$  
*Sequence transitive closure* of sequence $s$ is a sequence containing $s$ along with some new members appended to $s$ at the end. Each new member is implied through a transitive relation, $\text{TranSeq}_s(x, y)$, which starts from some where in sequence $s$ and moves only forward in $s$.

$\text{TranSeq}_s(x, y)$  
Given sequence $s$ and $s^\circ$, for each member of $s^\circ$, $(x, y)$, there exists a sequence, *transitive sequence* ($\text{TranSeq}_s(x, y)$), which is subsequence of $s$, and implies that member.

$\text{FirstPair}_s(x, y)$  
Given sequence $s$ and considering $(x, y) \in s^\circ$ and $\text{TranSeq}_s(x, y) = ((u_0, v_0), (u_1, v_1), \ldots, (u_n, v_n))$, then *First Relation in Transitive Sequence for* $(x, y)$, $\text{FirstPair}_s(x, y)$, is $(u_0, v_0)$.

$s \boxplus_i v$  
Inserts value $v$ in sequence $s$ at position $i$.

$\mathcal{V}_{F'} \subseteq \mathcal{V}_F$  
A dependency rule in EPD $F'$, $\mathcal{V}_{F'}$, respects a dependency rule in EPD $F$, $\mathcal{V}_F$, if $\mathcal{V}_{F'}$ does not introduce more dependency pairs on $F$ ports than $\mathcal{V}_F$ itself does. Furthermore, $\mathcal{V}_{F'}$ should observe the sequence of dependency pairs in $\mathcal{V}_F$.

$\text{AltTran}(r, s)$  
*Alternative Transitive closure* of two sequences of dependency pairs $r$ and $s$, is a dependency pair $(x, y)$ that is created through a transitive relation which alternates between $r$ and $s$ members. Furthermore, starting from $r$ and $s$, the dependency pairs only happen by moving forward through the two sequences.
AltNonTranSeq(r, s) non transitive sequence of relations of two sequences of dependency pairs, r and s, is a binary operation resulting an alternating sequence of dependency pairs which are subsequences of the two sequences r and s, and are disjoint from AltTran(r, s), are positioned between the members of AltTran(r, s).

B.2 Chapter 3 Symbols

Shared(P, Q) Given two IAs, P and Q, their set of shared actions is the intersection of their actions.

SimpShared(P, Q) Given two PIAs, P and Q, their set of simple shared actions is the intersection of their simple actions.

Illegal(P, Q) The set of illegal states for two composable IAs or PIAs, P and Q, is a subset of the product of states in P and Q which have an output action belonging to Shared(P, Q) enabled while the corresponding input action is not enabled in that state.

InputSchedule(s) Considering s as a sequence of actions, InputSchedule(s) represents a sequence of entirely input actions which is subsequence of actions in s.

OutputSchedule(s) Considering s as a sequence of actions, OutputSchedule(s) represents a sequence of only output actions which is subsequence of actions in s.
**InternalSchedule**\((s)\) Considering \(s\) as a sequence of actions, \(\text{InternalSchedule}(s)\) represents a sequence of only internal actions which is subsequence of actions in \(s\).

\(P \otimes Q\) Product of two IAs or PIAs, \(P\) and \(Q\), is a IA or PIA, respectively. It consists of the sets of interleaved states and steps of \(P\) and \(Q\). Furthermore, at each state all possible actions are enabled. It is possible to have illegal states in the product of two IAs or PIAs.

\(\varepsilon\text{-closure}_P(v)\) For a state \(v\) in an IA \(P\), \(\varepsilon\text{-closure}_P(v)\), is a set containing \(v\) and all states that can be reached from \(v\) through internal steps. For PIAs, \(\varepsilon\text{-closure}_P(v)\) is defined only for abstract states and contains all reachable abstract states reachable through simple internal actions or complex actions belonging to \(\lambda^H_P\).

\(\text{ExtEn}^I_P(v)\) For a state \(v\) in an IA \(P\), the set of *externally enabled input actions* is the set of input actions that are enabled in all states belonging to \(\varepsilon\text{-closure}_P(v)\).

\(\text{ExtEn}^O_P(v)\) For a state \(v\) in an IA \(P\), the set of *externally enabled output actions* is the set of output actions that are enabled in at least one state of \(\varepsilon\text{-closure}_P(v)\).

**ExtDest**\(_P(v,a)\) For a state \(v\) and an externally enabled action, \(a \in \text{ExtEn}^I_P(v) \cup \text{ExtEn}^O_P(v)\), *externally reachable states* are the states that can be reached by \(a\) from any of states belonging to \(\varepsilon\text{-closure}_P(v)\).
\( V_{comp} \) or complex states: The set of complex states in \( P \), a PIA, contains states which are used to define constituent steps of complex actions. By interleaving states of two PIAs, those resulting states that have at least one complex state are considered complex state of the interleaving.

\( \lambda_{I}^{P} \): For \( P \), a PIA, the set of input complex actions is subset of \( A_{comp}^{P} \) and has a schedule with at least one simple input action.

\( \lambda_{O}^{P} \): For \( P \), a PIA, the set of output complex actions is subset of \( A_{comp}^{P} \) and has a schedule with at least one simple output action.

\( \lambda_{H}^{P} \): For \( P \), a PIA, the set of internal complex actions is subset of \( A_{comp}^{P} \) and has a schedule entirely consisting of simple internal actions.

\( x \sqsubseteq y \): Given two execution fragments, \( x \) and \( y \), \( x \) is subfragment of \( y \), if \( sched(x) \subseteq sched(y) \).

IllegalComp(\( P, Q \)): Considering two composable PIAs, \( P \) and \( Q \), and their set of interleaved states \( V_{P \oplus Q} \), any complex state \( (p, q) \in V_{P \oplus Q} \) that does not initiate a step, belongs to the set of illegal complex states of \( P \) and \( Q \).

CompShared(\( P, Q \)): Given two composable PIAs, \( P \) and \( Q \), their set of complex shared actions is a set of complex actions belonging to \( P, Q \) or both, where for each of such complex actions, there exists at least one complex action in the other PIA such that at least there is one simple action common between the two PIAs.
| **Embedded** *P, Q* | Given two PIAs, *P* and *Q*, their set of embedded complex actions is the set of complex actions belonging to *P*, *Q* or both. Schedule of each of such complex actions must be subsequence of the schedule of at least one complex action in the other. PIA. |
| **Embedding** *P, Q* | Given two PIAs, *P* and *Q*, their set of embedding complex actions is the set of complex actions belonging to *P*, *Q* or both. For each of such complex actions, suppose it is *A*, there must exist a complex action, in the other PIA, such that its schedule is subsequence of sched(*A*). |
| **V** *P* ⋇ *Q* | The set of interleaved states for two PIAs, *P* and *Q*, is the cartesian product of their states. |
| **τ** *P* ⋇ *Q* | The set of interleaved steps for two PIAs, *P* and *Q*, is the set of steps which can be possibly enabled at **V** *P* ⋇ *Q*, and also respect the sequentiality of complex actions in *P* and *Q*. |
| **V** *P* ⋈ *Q* | The set of legal interleaved states is equal to **V** *P* ⋇ *Q* minus IllegalComp(*P, Q*). |
| **τ** *P* ⋈ *Q* | The set of legal interleaved steps consists of only those steps in **τ** *P* ⋇ *Q* that are not defined over IllegalComp(*P, Q*). |
| **Δ** (*v, u*) | For each complex action belonging to the set of legal interleaved states, **Δ** (*v, u*) determines the execution fragment that it belongs to. |
**B.2. CHAPTER 3 SYMBOLS**

\[ \chi_{P \otimes Q} \]
Considering all execution fragments representing all complex steps in \( P \otimes Q \), the set \( \chi_{P \otimes Q} \) represents the first step in such execution fragments.

\[ \sigma_{P \otimes Q} \]
Considering all execution fragments representing all complex steps in \( P \otimes Q \), the set \( \sigma_{P \otimes Q} \) represents the first complex state in such execution fragments.

\[ \Psi_{P \otimes Q}(r, s) \]
Given a state \((r, s) \in V_{P \otimes Q}\) which belongs to \( \sigma_{P \otimes Q} \), \( \Psi_{P \otimes Q}(r, s) \) maps that state to exactly one step in \( P \) or \( Q \). Such a step has an execution fragment with an schedule equal to \( \Delta(r, s) \).

\[ ExtEn_{P}^{I,simp}(v) \]
For an abstract state \( v \) in PIA \( P \), the set of externally enabled simple input actions is the set of input actions that are enabled in all states belonging to \( \varepsilon\text{-}closure_{P}(v) \).

\[ ExtEn_{P}^{O,simp}(v) \]
For an abstract state \( v \) in PIA \( P \), the set of externally enabled simple output actions is the set of output actions that are enabled in at least one state belonging to \( \varepsilon\text{-}closure_{P}(v) \).

\[ ExtEn_{P}^{I,comp}(v) \]
For an abstract state \( v \) in PIA \( P \), the set of externally enabled complex input actions is the set of complex actions belonging to \( \lambda_{P}^{I} \) such that they are enabled in all states belonging to \( \varepsilon\text{-}closure_{P}(v) \).

\[ ExtEn_{P}^{O,comp}(v) \]
For an abstract state \( v \) in PIA \( P \), the set of externally enabled complex output actions is the set of complex actions belonging to \( \lambda_{P}^{O} \) such that they are enabled in at least one state belonging to \( \varepsilon\text{-}closure_{P}(v) \).
\( ExtDest_P(v, a) \) For an abstract state \( v \) of PIA \( P \) and an externally enabled action, \( a \in (ExtEn^I_P(v) \cup ExtEn^O_P(v) \cup ExtEn^I_{comp}(v) \cup ExtEn^O_{comp}(v)) \), externally reachable states are the states that can be reached by \( a \) from any of states belonging to \( \varepsilon\)-closure\(_P(v)\).
Bibliography


