CORE
Metadata, citation and similar papers at core.ac.uk
Provided by AMS Tesi di Dottorato

Dottorato di Ricerca in Informatica
Università di Bologna e Padova
INF/01 INFORMATICA

Data and Behavioral Contracts for Web Services

Samuele Carpineti

March 2007

Coordinatore:
Prof. Özalp Babaoğlu

Tutore:
Prof. Cosimo Laneve

Abstract

The recent trend in Web services is fostering a computing scenario where loosely coupled parties interact in a distributed and dynamic environment. Such interactions are sequences of XML messages and in order to assemble parties – either statically or dynamically – it is important to verify that the "contracts" of the parties are "compatible".

The Web Service Description Language (WSDL) is a standard used for describing one-way (asynchronous) and request/response (synchronous) interactions. Web Service Conversation Language extends WSCL contracts by allowing the description of arbitrary, possibly cyclic sequences of exchanged messages between communicating parties. Unfortunately, neither WSDL nor WSCL can effectively define a notion of compatibility, for the very simple reason that they do not provide any formal characterization of their contract languages.

We define two contract languages for Web services. The first one is a data contract language and allow us to describe a Web service in terms of messages (XML documents) that can be sent or received. The second one is a behavioral contract language and allow us to give an abstract definition of the Web service conversation protocol. Both these languages are equipped with a sort of "sub-typing" relation and, therefore, they are suitable to be used for querying Web services repositories. In particular a query for a service compatible with a given contract may safely return services with "greater" contract.

Contents

Abstract iii

List of Figures ix

I Introduction 1

1 Introduction 3

1.1 Web services . . . 3

1.2 Web services and contracts . . . 5

1.3 Structure of the thesis . . . 7

II Data contracts 11

2 The Schema Language 13

2.1 Introduction . . . 13

2.2 Related works . . . 16

2.3 Schemas with references . . . 17

2.4 The subschema relation . . . 21

2.5 Primitive types . . . 28

2.6 The algorithmic subschema . . . 29

2.6.1 Properties of the algorithmic subschema 31
2.7 Label-determined schemas 34
2.7.1 The code and its computational complexity 37
2.8 Conclusion 39

3 Encoding of WSDL interfaces 43
3.1 Introduction 43
3.2 WSDL interfaces 46
3.2.1 Syntax of WSDL interface definitions 47
3.2.2 Encoding of WSDL interfaces 50
3.3 XML-SCHEMA 54
3.3.1 Syntax of XML-SCHEMA 56
3.3.2 Encoding of XML-SCHEMA 57
3.4 Conclusions 60

III Behavioral contracts 63

4 The Finite Contract Language 65
4.1 Introduction 65
4.2 Related work 67
4.3 The finite contract language 68
4.3.1 Subcontract relation and dual contracts 69
4.3.2 Contract compliance 75
4.3.3 Contract structural equivalence 76
4.4 Message exchange patterns in WSDL 77
4.5 Process Compliance 79

5 The Recursive Contract Language **85**

5.1 Recursive contracts . . . 86

5.1.1 Subcontract relation . . . 87

5.1.2 Contract compliance . . . 95

5.2 Conversations in WSCL . . . 97

5.3 Process Compliance . . . 99

5.4 Process Estimations . . . 105

5.4.1 Contracts and processes . . . 105

5.4.2 Underestimation . . . 109

5.4.3 Overestimation . . . 112

5.5 Regular Processes . . . 115

IV Conclusions **119**

6 Conclusions **121**

References **125**

List of Figures

1.1 The Web service stack. 4

3.1 Interaction with a UDDI registry. 44
3.2 The structure of a WSDL. 45

5.1 Contract of a simple e-commerce service as a WSCL diagram. 97
5.2 Contract of a e-commerce service as a WSCL diagram with cycles. 100

Part I

Introduction

Chapter 1

Introduction

1.1 Web services

Initially, most middleware platforms were designed to work on a single Local Area Network (LAN). As the number of LANs within companies started to grow, and different branches of the same company implemented their own middleware-based systems, the need arose for different middleware platforms to communicate with each other. When such interactions take place across company boundaries, they are called businnes-to-businnes exchanges and their purpose is to fully automate interactions between companies. Technically, implementing this kind of interaction requires the ability to invoke services residing in a different company. Conceptually, this is relatively easy to do, and thus there is a strong temptation to implement Internet-wide integration using a simple extension of conventional communication protocols (as CORBA [Gro04a] does). The problem is that assumptions underlying this rarely hold when the interaction occurs across the Internet. For example companies may be protected by firewalls which impose significant restrictions on communications. Furthermore components must agree on interface definitions and data formats to be used by the two applications. Finally, a directory server is required to enable service discovery, and, especially in inter-organizational settings, it may not be clear where this server should reside and who should manage it. These problems are a big part of what Web services try to solve.

Web services technology is a way to expose the functionality of an information system and make it available through Web standards. Web services and their technologies are

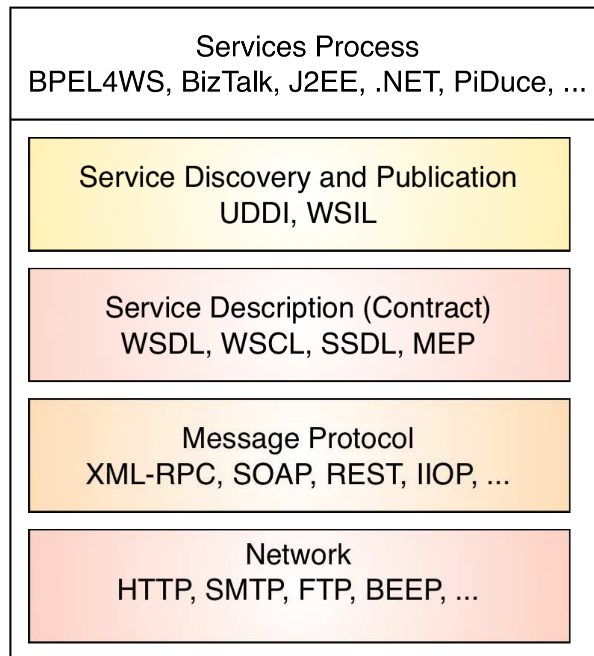


Figure 1.1: The Web service stack.

being developed with one specific use in mind: that of being entry point to the local information system. Thus, the primary use of a Web service is that of exposing – through the Web service interface – the functionality performed by internal systems and making it discoverable and accessible through the Web in a controlled manner. Web services are therefore analogous to sophisticated *wrappers* that encapsulate one or more applications by providing a unique interface and a Web access. The use of standard technologies is a key point because it allows to cope with heterogeneity, facilitating integration between different applications. In particular Web services are based on standards built over XML that, in the last ten years, has become a universally used meta-language for platform independent data representation.

The W3C [Gro04d](the Web Service Activity Group) defines a Web service as a "*software system identified by a URI, whose public interfaces and bindings are defined and described using XML. Its definition can be discovered by other software systems. These systems may then interact with the Web service in a manner prescribed by its definition, using XML based messages conveyed by Internet protocols*". The definition hints at how Web services should work. Web services must be able of being *defined*, *described*, *in-*

voked, and *discovered* in a way that can be automatically processed by other machines. Although these problems were addressed for conventional middleware, solutions have been reconsidered for Web services that are autonomous and loosely coupled systems. Having homogeneous components considerably reduces the difficulties of integration. This is also true for Web services, which are indeed wrappers and are homogeneous, in the sense that they interoperate through Web standards. As such, Web services constitute the base on which we can construct middleware supporting application integration on the Web by allowing designers to avoid the problems generated by the lack of standardization typical of previous approaches. Actually, what generated the need for Web services, is essentially the necessity of facilitating B2B integrations.

The Web service stack and some of the related standard is shown in Figure 1.1. Invocations are performed over standard network protocols – HTTP, SMTP, FTP, BEEP – sending XML messages packaged using one messaging protocol such as SOAP [Gro03], REST, IIOP and XML-RPC. The description of the service is base on standardized XML oriented languages – WSDL [Gro, BL06], WSCL [BBB+02], SSDL [SP], MEP [CHL+06] – and service discovery and service publication are also standardized in UDDI [Spe02] and WSIL [BBM+01]. Service process – based on several architecture J2EE [Sun06], .NET [Mic06], PIDUCE [CLP06], BPEL [CGK+02] – support this stack in a way that it is transparent to the user.

1.2 Web services and contracts

The recent trend in Web services is fostering a computing scenario where loosely coupled parties interact in a distributed and dynamic environment. Such interactions are typically sequences of messages that are exchanged between parties. The environment, being dynamic, makes it not always feasible to define or assemble parties statically. In this context, it is fundamental for clients to be able to search, also at run-time, services with the required capabilities, namely the format of the exchanged messages, and the protocol – or *contract* – required to interact successfully with the service. In turn, services are required to publish such capabilities in some known repository. Thereafter clients may query the repository asking for servers *compatible* with a certain contract. Internet industry leaders such as Amazon, eBay, Google, and FedEx offer publicly-available Web services as APIs into desirable data and familiar functionality: Amazon offers Web services for its product search,

cart system, and wish lists, eBay Web services expose item search, bidding, and auction creation, Google Web services offer access to Web search, cache, and spelling applications, and FedEx Web services connect to its package shipping and tracking systems. Such functionalities are described by public interfaces that can be interpreted by humans and other Web services. Such interfaces can be also published in public repositories. The Web Service Description Language (WSDL) [Gro, CMRW06, CHL+06] provides a standardized technology for describing the interface exposed by a service. Such description includes the service location, the format (or *schema*) of the exchanged messages, the transfer mechanism to be used (i.e. SOAP-RPC, or others), and the *data contract*. The data contract defines the set of XML documents accepted by the service. Traditional programming language do not provide a native support to XML therefore they cannot prevent – statically – run-time type errors during invocations. For instance, a JAVA [Mic05] program interacting with a Web service operation may be written in two ways. The first way is to manually build the string containing the XML and send it to the server. The second way is to use some tool for importing the service. Such tools map the XML document that can be sent into a tree of JAVA classes. The programmer may use the available methods for setting the value of the fields. When the service operation is invoked such class is serialized into an XML bytestream representing the request. Both the approaches are not very satisfactory. In first case it is easy to make errors while building the XML document. In the second case the gap between the two type systems (XML schemas and classes) prevents only some run-times errors. For these kind of reasons, in these years, some programming languages for XML processing have been proposed [ACE+04, Com04, HP03, BCF03, Cha03, ACJ04, GLPS04]. The contract language we propose is an extension of one of such a the type system – the one used in XDUCE – with Web service references and it has been prototyped in PIDUCE [BLM05, CLP06, CLM05] – a programming language for experimenting Web service related technologies.

WSDL contracts are basically limited to one-way (asynchronous) and request/response (synchronous) interactions. For instance in a WSDL interface it is not possible to specify that an operation must be called before another operation. Therefore WSDL interfaces do not currently address the problem of how a service can define the sequences of legal message exchanges it supports. An attempt of specifying this *behavioral contract* is the Web Service Conversation Language (WSCL) [BBB+02]. It extends WSDL contracts by allow-

ing the description of arbitrary, possibly cyclic sequences of exchanged messages between communicating parties. Thus, WSCL defines the conversation protocol implemented by the service. Both WSDL and WSCL documents can be published in repositories [BKL01, CJ04] so that they can be searched and queried.

The use of a contract in a repository immediately poses an issue related to the *compatibility* between different published contracts. It is necessary to define precise notions of contract similarity and compatibility and use them to perform service discovery in the same way as, say, type isomorphisms are used to perform library searches [Rit93, Cos95]. Unfortunately, neither WSDL nor WSCL can effectively define these notions, for the very simple reason that they do not provide any formal characterization of their contract languages. This cries out for a mathematical foundation of contracts and the formal relationship between clients and contracts.

In this thesis we define two contract languages for Web services. The first one is a data contract language that allows us to describe a Web service in terms of messages (XML documents) that can be sent or received. The second contract language is a behavioral contract language that allows us to give an abstract definition of the Web service conversation protocol. Both these languages are equipped with a *compatibility* relation so that they are suitable to be used for querying Web services repositories. A query for a service compatible with a given contract may safely return services with a *greater* contract.

1.3 Structure of the thesis

Chapter 2 In Chapter 2, that is an extended version of the conference paper at ESOP'06 [CL06], we design a data contract language – a schema language – for representing XML documents containing references to remote services. Such a language is equipped with a subschema relation that allows us to verify whenever a services is used according to its contract: sending and receiving the proper data. Since the subschema relation computes tree language containment, it turns out to be computationally expensive. To avoid run-time degradations of Web services technologies, we impose a language restriction to diminish the cost of the subschema relation.

Chapter 3 In Chapter 3 the expressiveness of the schema language we have designed in Chapter 2 is compared with the two standards used for describing Web service operations:

XML-SCHEMA and WSDL. In particular we show how to encode interfaces written using these two standards in our schema language.

Chapter 4 In Chapter 4, that extends the conference paper at WSFM'06 [CCLP06], we define a calculus for behavioral contracts along with a subcontract relation, and we formalize the relationship between contracts and processes (that is clients and services) exposing a given contract. Contracts are made of actions to be interpreted as either message types or communication ports. Actions may be combined by means of two choice operators: $+$ represents the *external choice*, meaning that the interacting part decides which one of alternative conversations to carry on; $\oplus$ represents the *internal choice*, meaning that the choice is not left to the interacting part. Then we devise a *compliance relation* that can be used for querying Web services repositories: a query for a service with a given contract may "safely" return services with complying contract. By safely we mean that the client will complete its protocol even if the interaction may result into unused interaction capabilities on the server side. We then extrapolate contracts out of processes, that are a recursion-free fragment of CCS. We finally demonstrate that a client completes its interactions with a service provided the corresponding contracts comply. The expressiveness of our contract language is gauged by encoding WSDL message exchange patterns into our contract language. Because of the $\preceq$ relation between contracts, we are able to draw some interesting considerations about similar exchange patterns, and order them according to the client's needs.

Chapter 5 In Chapter 5 we extend the contract language of Chapter 4 with recursive contracts. We lift the notions of subcontract and compliance to recursive contracts and we show how to extrapolate contracts out of *regular* CCS processes. As regards non-regular processes two relations – called *overestimation* and *underestimation* – are defined. Overestimation is used to approximate the conversation protocol implemented by clients with a "more demanding" protocol. Similarly, underestimation is used to approximate the conversation protocol implemented by (non-regular) services with a "simpler" protocol. We then verify that, whenever a client is overestimated with a contract σ and a server is overestimated with a contract σ' such that σ and σ' comply, then the two processes also comply. For this reason overestimation can be used by clients for querying repositories and underestimation can be used by service providers for publishing their contracts.

In the recursive case, to show the expressiveness of our recursive contract language, an encoding of two WSCL conversations is given. As for WSDL message exchange pattern, thanks to the $\preceq$ relation between contracts, we are able to order WSCL conversations with respect to the client's needs.

Chapter 6 Chapter 6 draws our conclusion and hints at some future works.

Part II

Data contracts

Chapter 2

The Schema Language

In this chapter we design a basic schema language for describing XML documents with references to Web services and to their operations. The assessment that references are used according to their capabilities is given by a subschema relation. We give two definitions of our subschema relation: one as a form of schema simulation, defined coinductively, and one by means of inference rules. Then we prove that the algorithm derived from the inference rules is sound and complete with respect to the coinductive definition. Since the subschema relation turns out to be computationally expensive and this may be problematic in the Web service context where the relation is often used at runtime, we study a restriction of the schema language to reduce the cost of the subschema relation.

Structure of the chapter. Section 2.1 introduces XML and XML schema languages. Section 2.2 refers to some related work; Section 2.3 presents the schema language and Section 2.4 defines the subschema relation. Section 2.5 shows an extension of the schema language with base types. Section 2.6 defines the algorithmic version of the subschema relation and proves its correctness. Section 2.7 studies a restriction of the schema language having a polynomial subschema relation. Section 2.8 concludes with some final remarks.

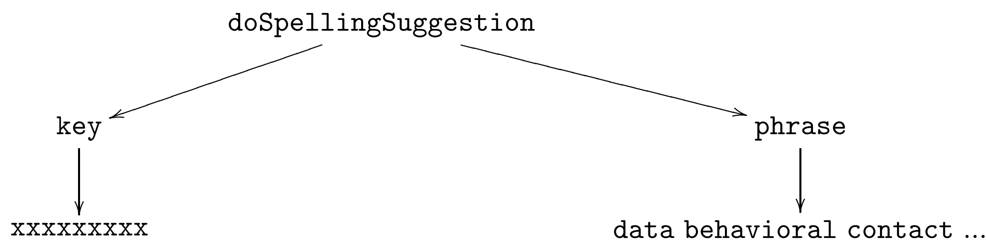
2.1 Introduction

The Extensible Markup Language XML [Gro04f] is a concrete syntax used for representing abstract trees in textual form. XML trees contains nodes called *elements* that are labeled

by a tag (a name), and that may contain either characters or an arbitrary number of children. An example of XML document is:

```
< doSpellingSuggestion >
 < key > xxxxxxxxx < /key >
 < phrase > data behavioral contact web services < /phrase >
< /doSpellingSuggestion >
```

where $< \dots >$ and $< / \dots >$ define the beginning and the end of an element respectively (intentionally we wrote `contact` instead of `contract`). The main constraint in writing XML document is that they have to be well-formed. This means that if an element starts in the content of an element, it also finishes in the content of the same element. Said otherwise, elements, delimited by start and end tags, nest properly within each other. It is this constraint that allows to view an XML document as a tree. For instance the XML above is represented by the following tree:



Since the choice of the element names is left to users, XML provides a syntactic foundation for the creation of custom markup languages through definition of *schemas*. Indeed, the abstract syntax of XML documents is rigid in the sense that they must adhere to the general rules of XML. This rigidity assures that all XML-aware software can parse and interpret the information within them. The schema specifies a set of additional constraints on the structure of the documents. Schemas may restrict element names and their content in such a way that only some elements with specific names and in a certain order and with a certain arity may appear.

The simplicity of XML and the fact that it is essentially a human readable and platform-independent representation of information are the main motivations for its use in business to business applications. In particular Web services are strongly based on XML-related technologies because, being standard, are guaranteed to be interoperable. Indeed, data exchanged between Web services are just XML documents. For instance the document above (< `doSpellingSuggestion` > ...) may be sent to the Google Web service located at

`http://api.google.com/search/beta2` receiving back another XML document as:

< return > data behavioral contract web services < /return >

containing the spelling suggestion for the sent phrase. In our notation – borrowed from Hosoya et al. [HVP00a] – we say that the Google Web service accepts documents of schema

doSpellingSuggestion[key[String],phrase[String]]

and returns documents of schema return[String].

Web services technologies also require the possibility to express and communicate references to remote services [Vin04, GNO03, Gro04e]. In facts, these requirements are recognized in the new specification of the Web Service Description Language (WSDL) [BL06], which extends the schemas with Web service references. This is a must in many business to business applications where interactions are typically asynchronous or in batch mode. Rather than direct invocations, requests are batched and routed through queues and responses are handled in the same way (systems based on EDIFACT [Uni] and SWIFT [fWIFT] are typical loosely coupled architectures working in this way). However, the implementation of asynchronous interactions require, at least, the communication of the callback address. These asynchronous interactions are also used in other settings. For instance if the followingXML document

< appID > firefox < /appID >
< query > wsdl < /query >
< region > it < /region >
< language > it < /language >
< output > json < /output >
< callback > http : //www.foo.com/service/format < /callback >

is sent to the Yahoo Web service, the result of the search of the word query will be sent to the service which address specified by the callback address (a URI) corresponding to the formatting service. In our notation the value above may have the schema

appId[String],query[String],region[String],
language[String],output[String],$\langle \texttt{Result} \rangle^{0}$

meaning that the value is a sequence finishing with a reference that must be used by the server only for outputting document of schema Result (the syntax of reference has

been borrowed from Pierce and Sangiorgi [PS93]). This is not enough for describing Web services that are logical grouping of operations and not just single operations. For instance the Yahoo interface allow – as callback address – a service with two operations `result` and `noresult`, one to be called whenever there are some result and another one to be called when no results are available. In this case the schema of the reference may be $\{\texttt{result}:\langle\texttt{Result}\rangle^{\texttt{O}}\,\texttt{noresult}:\langle\texttt{()}\rangle^{\texttt{O}}\}$.

The communication of references requires the ability to verify that the receiver uses them according to their contract: sending proper data and performing the permitted operations. Therefore a mechanism for comparing schemas with references should be provided. For instance a client must be allowed to send to the Yahoo Web service a callback which is able to process, not only documents with schema `Result`, but a larger set of documents. Similarly, the client must be allowed to send to the Yahoo Web service a reference to a service which provide more operations (instead of `result` and `noresult` only). This is what we call subschema relation. In a nutshell the subschema relation allows to compare schemas in terms of XML documents they describe. As in [HVP00a] it reduces to tree set containment when values do non contain references and therefore we claim that it is appropriate for describing XML documents exchanged between Web services.

2.2 Related works

Several schema languages have been recently proposed for describing the tree-structure of XML documents. We recall DTD [Gro04f] and XML-SCHEMA [Gro04c] proposed by the W3C, RELAX-NG [CM01] by Clark and Murata, and XDUCE regular expression types [HVP00b] by Hosoya and Pierce. We refer to [MLM01] for an analysis of their expressiveness.

These schema languages, and specifically XML-SCHEMA, are used in WSDL [Gro, BL06] documents that are interfaces of Web services describing the messages sent and/or received by the services and the information for reaching the services (location, transport protocol, etc.). However the new specification of WSDL [BL06], that is now a W3C Proposed Recommendation, allows to describe Web services exchanging documents containing typed references. It stands that schema languages need to be extended in order to deal with such reference values.

The schema language studied in this chapter is an extension of XDUCE types with

references and is similar to those introduced in languages extending pi-calculus with XML datatypes [AB05, BLM05, CNV05]. The design of the schema language of [BLM05] has been strongly affected by this study. As a minor difference, reference schemas – called channel schemas in [BLM05] – only have output capabilities and record schemas have not been introduced. The schema language of [AB05] includes abstractions, that we do not consider, and for the rest is simpler than the one we present here. In particular the subschema relation is not as powerful as the one we define. For instance, unions do not distribute over sequences – $S\,,S' + S''$ is not subschema of $S\,,S' + S\,,S''$ – and over labelled constructors – $a[b[\,] + c[\,]\,]$ is not subschema $a[b[\,]\,] + a[c[\,]\,]$.

The types in [CNV05] include channels with capabilities, union, product, intersection and negation. The definition of subschema is semantic, by means of a set-inclusion on a set-theoretic model. Our schema language is simpler than [CNV05] and the notion of subschema is quite different. For example, in our case, top and bottom are derived schemas and channel schemas may be nested at wish, while this is problematic in presence of recursion and intersection. The contribution [CNV05] overlooks the restrictions for reducing the computational complexity of the subschema relation that turns out to be hyperexponential.

2.3 Schemas with references

We use three disjoint countably infinite sets: the *tags*, ranged over by $a, b, \ldots \in \mathcal{L}$, the *fields* ranged over by $\mathtt{m}_1, \mathtt{m}_2 \ldots \in \mathcal{M}$, and the *schema names*, ranged over by $\mathtt{U}, \mathtt{V}, \ldots \in \mathcal{N}$. The term κ is used to range over I, O, and IO. The syntax of our language includes the categories of *labels* and *schemas* defined by the rules in Table 2.1.

Labels. Labels specify collections of tags. The semantics of labels is defined by the following function $\widehat{\cdot}$:

$$\widehat{a} = \{a\} \qquad \widehat{\sim} = \mathcal{L} \qquad \widehat{L + L'} = \widehat{L} \cup \widehat{L'} \qquad \widehat{L \setminus L'} = \widehat{L} \setminus \widehat{L'}$$

($\sim$ represents the whole sets of tags). The intersection operator on labels is derived in terms of difference and union as: $L \cap L' \stackrel{\text{def}}{=} \sim \setminus ((\sim \setminus L) + (\sim \setminus L'))$.

Table 2.1: The schema language

L ::=		**label**	S ::=		**schema**
	a	(tag)		$\texttt{()}$	(void schema)
	~	(any label)		$\langle S\rangle^\kappa$	(reference schema)
	$L + L$	(union)		$\{(\texttt{m}_h{:}S_h)^{h\in H}\}$	(record schema)
	$L \setminus L$	(difference)		$L[S]$	(labelled schema)
				$S\texttt{,}S$	(sequence schema)
				$S + S$	(union schema)
				$\texttt{U}$	(schema name)

Schemas. Schemas describe (XML) documents that are structurally similar. The schema $\texttt{()}$ describes the empty document. The schema $\langle S\rangle^\kappa$ describes references (channels) that carry messages of schema S and that may be used with *capability* $\kappa \in \{\texttt{I}, \texttt{O}, \texttt{IO}\}$. The capabilities $\texttt{I}, \texttt{O}, \texttt{IO}$ mean that the channel can be used for performing inputs, outputs, and both inputs and outputs, respectively. For example $\langle\texttt{Int}\rangle^{\texttt{O}}$ describes the set of channel literals that may be invoked with integers. The schema $\{(\texttt{m}_h{:}S_h)^{h\in H}\}$ describes a record with fields $\texttt{m}_h$ and schema S_h. We assume fields to have different names and H to be finite. In the following we identify records up to fields commutativity (e.g. $\{\texttt{m}_1 : S_1\ \texttt{m}_2 : S_2\} = \{\texttt{m}_2 : S_2\ \texttt{m}_1 : S_1\}$). The schema $L[S]$ describes documents made of an element with a tag in L and containing a document of schema S. The schema $S\texttt{,}S'$ describes sequences starting with a document of schema S and finishing with a document of schema S'. In what follows we may omit $\texttt{()}$ appearing in sequences. Therefore $L[\texttt{()}]$, $\texttt{()}\texttt{,}S$ and $S\texttt{,}\texttt{()}$ are shortened into $L[\,]$, S, and S, respectively. The schema $S+S'$ describes the – possibly non-disjoint – set of documents belonging to either S or to S'. In what follows we assume the following precedence between operators: "$\texttt{,}$">"+".

Schemas include schema names that are bound by *finite* maps $\mathcal{E}$ from schema names to schemas such that, for every $\texttt{U} \in \texttt{dom}(\mathcal{E})$, the schema names in $\mathcal{E}(\texttt{U})$ belong to $\texttt{dom}(\mathcal{E})$. Maps $\mathcal{E}$ are *well-formed* according to the definition below. Let $\texttt{tls}(S)$ – the set of the top level names in S – be the least set of schema names such that:

$$\texttt{tls}(S) = \begin{cases} \{\texttt{U}\} \cup \texttt{tls}(\mathcal{E}(\texttt{U})) & \text{if } S = \texttt{U} \\ \texttt{tls}(T) \cup \texttt{tls}(T') & \text{if } S = T + T' \text{ or } S = T\texttt{,}T' \\ \emptyset & \text{otherwise} \end{cases}$$

Let also $\texttt{ntail}(S)$ be the least set such that:

$$\texttt{ntail}(S) = \begin{cases} \{\texttt{U}\} \cup \texttt{tls}(\mathcal{E}(\texttt{U})) & \text{if } S = \texttt{U} \\ \texttt{ntail}(T) \cup \texttt{ntail}(T') & \text{if } S = T + T' \\ \texttt{tls}(T) \cup \texttt{ntail}(T') & \text{if } S = T\texttt{,}T' \\ \emptyset & \text{otherwise} \end{cases}$$

Then $\mathcal{E}$ is well-formed if, for every $\texttt{U} \in \texttt{dom}(\mathcal{E})$, $\texttt{U} \notin \texttt{ntail}(\mathcal{E}(\texttt{U}))$. This definition is an immediate adaptation of that in [HVP00a] and together with the finiteness of the domain of $\mathcal{E}$ guarantees that schemas only define *regular tree languages*. Such languages retain a decidable sub-language relation [CDG+97]. For instance the schema $\mathcal{E}(\texttt{U}) = a[\,]\texttt{,U,}b[\,] + \texttt{()}$ – describing the non regular tree language $a[\,]^n b[\,]^n$ – is invalid: $\texttt{U} \in \texttt{ntail}(\mathcal{E}(\texttt{U}))$. On the contrary, the schemas $\mathcal{E}(\texttt{V}) = a[b[\,]\texttt{,V,}c[\,]\,] + \texttt{()}$ and $\mathcal{E}(\texttt{U}) = \texttt{V,U}$ are regular and valid.

It is worth to remark that *vertical* and *horizontal* recursion are provided by means of constant schema names. Then we can derive the "$*$" operator to describe arbitrary long sequences by means of a proper recursive definition. Indeed, $S^* = \texttt{U}$ with $\mathcal{E}(\texttt{U}) = S\texttt{,U} + \texttt{()}$.

Examples. We illustrate the syntax by means of few, but significant, sample schema name definitions. Let Bool, Blist, Btree, and Empty be such that

$$\begin{aligned} \mathcal{E}(\texttt{Bool}) &= \mathit{true}[\,] + \mathit{false}[\,] \\ \mathcal{E}(\texttt{Blist}) &= \texttt{Bool,Blist} + \texttt{()} \\ \mathcal{E}(\texttt{Btree}) &= \texttt{()} + \mathit{val}[\texttt{Bool}]\texttt{,}\mathit{left}[\texttt{Btree}]\texttt{,}\mathit{right}[\texttt{Btree}] \\ \mathcal{E}(\texttt{Empty}) &= a[\texttt{Empty}] \end{aligned}$$

The name Bool defines booleans that are encoded as tags *true* and *false* with an empty content (). The name Blist defines any flat sequence of labelled documents containing booleans; Btree defines documents that are binary trees of booleans. The name Empty defines an empty set of documents because this set is the least solution of the equation $\texttt{Empty} = a[\texttt{Empty}]$.

As regards references schemas, $\langle\texttt{Bool}\rangle^{\texttt{O}}$ describes references to operations that may be invoked with booleans; $\langle\texttt{Bool}\rangle^{\texttt{IO}}$ contains references that may be invoked with booleans *and* may receive notifications carrying booleans. The name NCbool defined as

$$\mathcal{E}(\texttt{NCbool}) = \langle\texttt{Bool}\rangle^{\texttt{O}} + \langle\texttt{NCbool}\rangle^{\texttt{O}}$$

describes the references to be invoked with booleans or with references to be invoked with booleans, etc., till some finite but not bound depth. (The nesting of channel constructors in [CNV05] is always bound.) We observe that a service querying a repository for references of schema $\langle\texttt{Bool}\rangle^{\texttt{O}}$ may get back a service of schema $\langle\texttt{Bool}\rangle^{\texttt{IO}}$ or of schema `NCbool`. Conversely, if the query is about references of schema $\langle\texttt{Bool}\rangle^{\texttt{IO}}$ then the repository will never return references of schema $\langle\texttt{Bool}\rangle^{\texttt{O}}$ nor `NCbool`. Indeed on $\langle\texttt{Bool}\rangle^{\texttt{O}}$ and `NCbool` input cannot be performed.

Record schemas may be used for describing references to remote services. For instance the name `TreeService` defined as

$$\mathcal{E}(\texttt{TreeService}) = \{\texttt{flatten}:\ \langle\texttt{Btree,}\langle\texttt{Blist}\rangle^{\texttt{O}}\rangle^{\texttt{O}}\ \texttt{tree}:\ \langle\texttt{Blist,}\langle\texttt{Btree}\rangle^{\texttt{O}}\rangle^{\texttt{O}}\}$$

describes a Web service with two operations: `flatten` and `tree`. The first one takes a `Btree` and a reference where the list of booleans it contains should be sent; the second one takes the list of booleans and the reference where the tree of booleans should be sent.

Remark. The subschema language without references and records is closed under union, difference, and intersection. This means that given two schemas we can define their union, difference and intersection in terms of another schema of our language [HVP00b]. Union closure is a consequence of the presence of *non-deterministic* union schemas. Then it is possible to define the union of two schemas even if they describe common values. For instance, XML-SCHEMA is not closed under union even if it provides for a union operator. Indeed, since union is limited to disjoint schemas $a[\texttt{Int}]+b[\texttt{String}]$ is allowed, but $a[\texttt{Int}]+a[\texttt{String}+\texttt{Int}]$ is not. Difference closure $S\backslash T$ follows by the fact that labels are represented as sets. For example $L[S]\texttt{,}S' \setminus L'[T]\texttt{,}T'$ is $(L \setminus L')[S]\texttt{,}S' + L[S \setminus T]\texttt{,}S' + L[S]\texttt{,}S' \setminus T'$. Intersection $S \cap T$ may be defined in terms of difference as $(S + T) \setminus (S \setminus T) \setminus (T \setminus S)$. This sublanguage has a decidable algorithm testing the emptiness of a schema. Thereafter $S <: T$ may be implemented as an emptiness test on $S \setminus T$. Channel schemas does not preserve the closures under difference: the schema $\langle a[\,]\rangle^{\texttt{O}} \setminus \langle b[\,]\rangle^{\texttt{O}}$ cannot be written in our schema language. For this reason these operators are primitive in [CNV05].

2.4 The subschema relation

The semantic definition of subschema in [HP03] does not adapt well to our language. In that paper, a language for *values* was introduced and a schema S was considered a subschema of T if the set of values described by S was contained in the set of values described by T. In our case values should contain references that do not carry any "structural" information about their schema. Therefore, in order to verify that a reference belongs to a schema S, we should verify the schema of the reference is a subschema of S. To circumvent this circularity we use an "operational" definition – a *simulation* relation – in the style of [AC93, PS93].

The subschema relation uses handles to manifest all the branches of the syntax tree of a schema.

Definition 2.1 (Handle) *Let* $S \downarrow R$*, read* S has handle R*, be the least relation over schemas such that*

$$
\begin{array}{ll}
\texttt{()} \downarrow \texttt{()} & \\
\langle S \rangle^{\kappa} \downarrow \langle S \rangle^{\kappa}\texttt{,()} & \\
\{(\mathtt{m}_h{:}S_h)^{h \in H}\} \downarrow \{(\mathtt{m}_h{:}S_h)^{h \in H}\}\texttt{,()} & \\
L[S] \downarrow L[S]\texttt{,()} & \text{if } L \neq \emptyset \text{ and, for some } R,\ S \downarrow R \\
S\texttt{,}S' \downarrow R & \text{if } S \downarrow \texttt{()} \text{ and } S' \downarrow R \\
S\texttt{,}S' \downarrow R\texttt{,}S' & \text{if } S \downarrow R \text{ and } R \neq \texttt{()} \text{ and, for some } \mathrm{R}',\ \mathrm{S}' \downarrow \mathrm{R}' \\
S + S' \downarrow R & \text{if } S \downarrow R \text{ or } S' \downarrow R \\
\mathtt{U} \downarrow R & \text{if } \mathcal{E}(\mathtt{U}) \downarrow R
\end{array}
$$

We observe that `Empty` has no handle. The schema $a[\ ]$,`Empty` has no handle as well; the reason is that a sequence has a handle provided that every element of the sequence has a handle. We also remark that channel schemas and service schemas always retains a handle. Let $handles(S) \stackrel{\text{def}}{=} \{R \mid S \downarrow R\}$. We say that a schema S is *not-empty* if and only if $\mathsf{handles}(S) \neq \emptyset$; it is *empty* otherwise.

Proposition 2.1 $\mathsf{handles}(S)$ *is finite.*

Proof: Let $h(S)$ be the function defined as

$$h(S)_{\mathcal{X}} = \begin{cases} 0 & \text{if } S \textit{ is empty} \\ 1 & \text{if } S = \texttt{()} \text{ or } S = \langle T \rangle^{\kappa} \text{ or } S = \{(\texttt{m}_h{:}S_h)^{h\in H}\} \\ 1 & \text{if } (S = L[T]\texttt{,}T' \text{ or } S = \{(\texttt{m}_h{:}S_h)^{h\in H}\}\texttt{,}T' \text{ or } S = \langle T \rangle^{\kappa}\texttt{,}T') \\ & \quad \text{and } S \textit{ is not-empty} \\ 0 & \text{if } S = \texttt{U} \text{ and } \texttt{U} \in \mathcal{X} \\ 1 + h(\mathcal{E}(\texttt{U}))_{\mathcal{X}\cup\{\texttt{U}\}} & \text{if } S = \texttt{U} \text{ and } \texttt{U} \notin \mathcal{X} \text{ and } S \textit{ is not-empty} \\ h(T)_{\mathcal{X}} + h(T')_{\mathcal{X}} & \text{if } S = T\texttt{,}T' \text{ and } S \textit{ is not-empty} \\ h(T)_{\mathcal{X}} + h(T')_{\mathcal{X}} & \text{if } S = T + T' \text{ and } S \textit{ is not-empty} \end{cases}$$

It is easy to verify that $h(S)_{\emptyset}$ is finite for every schema. The proof proceeds by induction on $h(S)_{\emptyset}$. The base case is obvious. The inductive case is by cases on the structure of S. We discuss the subcases $S = \texttt{U}$ and $S = S'\texttt{,}S''$. We observe that, by definition, $\mathsf{handles}(\texttt{U}) = \mathsf{handles}(\mathcal{E}(\texttt{U}))$. By inductive hypothesis $\mathsf{handles}(\mathcal{E}(\texttt{U}))$ is finite; therefore $\mathsf{handles}(\texttt{U})$ is finite as well. If $S = S'\texttt{,}S''$ then there are several subcases depending on the structure of S'. If $S' = \texttt{()}$ then $\mathsf{handles}(S) = \mathsf{handles}(S'')$ and we conclude by the inductive hypothesis. If S' is either a channel or a service or a labelled schema then we immediately conclude. If $S = \texttt{U}$ then $\mathsf{handles}(S) = \mathsf{handles}(\mathcal{E}(\texttt{U})\texttt{,}S'')$ and we conclude by the inductive hypothesis. $\square$

Definition 2.2 (Subschema) *Let $\leq$ be the least partial order on capabilities such that $\texttt{IO} \leq \texttt{I}$ and $\texttt{IO} \leq \texttt{O}$. A* subschema *$\mathcal{R}$ is a relation on schemas such that $S\ \mathcal{R}\ T$ implies:*

1. *$S \downarrow \texttt{()}$ implies $T \downarrow \texttt{()}$;*

2. *$S \downarrow \langle S' \rangle^{\kappa}\texttt{,}S''$ implies $T \downarrow \langle T_i \rangle^{\kappa_i}\texttt{,}T'_i$, for $1 \leq i \leq n$, with $\kappa \leq \kappa_i$, $S''\ \mathcal{R} \sum_{1\leq i\leq n} T'_i$, and, for every $1 \leq i \leq n$, one of the following conditions holds:*

   (a) either $\kappa_i = \texttt{O}$ and $T'_i\ \mathcal{R}\ S'$,

   (b) or $\kappa_i = \texttt{I}$ and $S'\ \mathcal{R}\ T'_i$,

   (c) or $\kappa_i = \texttt{IO}$ and $S'\ \mathcal{R}\ T'_i$ and $T'_i\ \mathcal{R}\ S'$;

3. *$S \downarrow \{(\texttt{m}_h{:}S_h)^{h\in H}\}\texttt{,}S'$ implies $T \downarrow \{(\texttt{m}_j{:}T_j)^{j\in J_i}\}\texttt{,}T'_i$, for $1 \leq i \leq n$, with $S'\ \mathcal{R} \sum_{1\leq i\leq n} T'_i$, and, for every $1 \leq i \leq n$, $J_i \subseteq H$, and, for every $j \in J_i$, $S_j\ \mathcal{R}\ T_j$;*

4. *$S \downarrow L[S']\texttt{,}S''$ then one of the following conditions holds:*

(a) *either* $T \downarrow L'[T']\,,T''$ *with* $\widehat{L} \cap \widehat{L'} \neq \emptyset$, $\widehat{L} \not\subseteq \widehat{L'}$, $(L \setminus L')[S']\,,S'' \;\mathcal{R}\; T$, *and* $(L \cap L')[S']\,,S'' \;\mathcal{R}\; T$;

(b) *or* $T \downarrow L_i[T_i]\,,T_i'$, *for* $1 \leq i \leq n$, *with* $\widehat{L} \subseteq \bigcap_{i\in\{1,\dots,n\}} \widehat{L_i}$ *and, for every* $J \subseteq \{1,\dots,n\}$, *either* $S' \;\mathcal{R}\; \sum_{i\in J} T_i$ *or* $S'' \;\mathcal{R}\; \sum_{i\in\{1,\dots,n\}\setminus J} T_i'$.

Let $<:$ *be the largest* subschema relation. *We write* $S \approx T$ *if* $S <: T$ *and* $T <: S$.

The definition of subschema is commented upon below.

Item 1 constraints greater schemas to manifest a void handle if the smaller one retains such a handle.

Item 2 deals with channel schemas $\langle S\rangle^\kappa\,,S'$. A set of handles $\langle T_i\rangle^{\kappa_i}\,,T_i'$ of the greater schema is selected such that S' is smaller than the union of the T_i''s and, in order to check the subschema relation between $\langle S\rangle^\kappa$ and $\langle T_i\rangle^{\kappa_i}$, the capability κ must be smaller than κ_i. Additionally, in case $\kappa_i = \mathtt{O}$ the subschema is inverted on the arguments (contravariance); in case $\kappa_i = \mathtt{I}$ the subschema is the same for the arguments (covariance), in case $\kappa_i = \mathtt{IO}$ the relation reduces to check the equivalence of the arguments (invariance). For instance we may verify the following relations

(i) $\langle a[\;] + b[\;]\rangle^{\mathtt{O}}\,,(c[\;] + d[\;]) <: \langle a[\;]\rangle^{\mathtt{O}}\,,(c[\;] + d[\;])$;

(ii) $\langle a[\;] + b[\;]\rangle^{\mathtt{O}}\,,(c[\;] + d[\;]) <: \langle a[\;]\rangle^{\mathtt{O}}\,,c[\;] + \langle b[\;]\rangle^{\mathtt{O}}\,,d[\;]$;

We note that, in the example (ii), since $c[\;] + d[\;] \not<: c[\;]$ and $c[\;] + d[\;] \not<: d[\;]$, both the handles of $\langle a[\;]\rangle^{\mathtt{O}}\,,c[\;] + \langle b[\;]\rangle^{\mathtt{O}}\,,d[\;]$ must be selected. Indeed a value with schema $\langle a[\;] + b[\;]\rangle^{\mathtt{O}}\,,(c[\;] + d[\;])$ is a sequence starting with a reference of schema $\langle a[\;] + b[\;]\rangle^{\mathtt{O}}$ and finishing with a document of either schema $c[\;]$ or schema $d[\;]$. Thus, if the tail has schema $c[\;]$ the value has schema $\langle a[\;] + b[\;]\rangle^{\mathtt{O}}\,,c[\;]$ and, by contravariance of $\langle\cdot\rangle^{\mathtt{O}}$, we conclude $\langle a[\;] + b[\;]\rangle^{\mathtt{O}}\,,c[\;] <: \langle a[\;]\rangle^{\mathtt{O}}\,,c[\;]$. By similar arguments, if the tail has schema $d[\;]$, the value has $\langle a[\;] + b[\;]\rangle^{\mathtt{O}}\,,d[\;]$ and we conclude $\langle a[\;] + b[\;]\rangle^{\mathtt{O}}\,,c[\;] <: \langle b[\;]\rangle^{\mathtt{O}}\,,c[\;]$. Said otherwise, unions distribute over sequences i.e. $S\,,(S' + S'') \approx S\,,S' + S\,,S''$ (cfr. rule (S-DIST-RCD) of [BP00]).

Item 3 deals with records. Similarly to item 2, a set of handles $\{(\mathtt{m}_j{:}T_j)^{j\in J_i}\}\,,T_i'$ of the greater schema is selected. This set must be such that S' is smaller than the union of the T_i''s. Furthermore the subschema relation between $\{(\mathtt{m}_h{:}S_h)^{h\in H}\}$ and $\{(\mathtt{m}_j{:}T_j)^{j\in J}\}$

is verified by checking that the operations provided by $\{(\mathtt{m}_j{:}T_j)^{j\in J}\}$ are also provided $\{(\mathtt{m}_h{:}S_h)^{h\in H}\}$ with a *compatible* schema. It follows that

$$\{\mathtt{flatten}:\langle\mathtt{Btree},\langle\mathtt{Blist}\rangle^{\mathtt{O}}\rangle^{\mathtt{O}}\ \mathtt{tree}:\langle\mathtt{Blist},\langle\mathtt{Btree}\rangle^{\mathtt{O}}\rangle^{\mathtt{O}}\} <: \{\mathtt{tree}:\langle\mathtt{Blist},\langle\mathtt{Btree}\rangle^{\mathtt{O}}\rangle^{\mathtt{O}}\}$$

$$\{\mathtt{flatten}:\langle\mathtt{Btree},\langle\mathtt{Blist}\rangle^{\mathtt{O}}\rangle^{\mathtt{O}}\} <: \{\mathtt{flatten}:\langle\mathtt{Btree}+\mathtt{Blist},\langle\mathtt{Blist}\rangle^{\mathtt{O}}\rangle^{\mathtt{O}}\}$$

Item 4 is the most involved one. It deals with handles $L[S']\mathtt{,}S''$. There are two cases: (a) either the greater schema has a handle $L'[T']\mathtt{,}T''$ with $\widehat{L}\cap\widehat{L'}\neq\emptyset$ and $\widehat{L}\not\subseteq\widehat{L'}$, or (b) the greater schema manifests a set of handles $L_i[T_i']\mathtt{,}T_i''$ with $\widehat{L}\subseteq\widehat{L_i}$. The case (a) accounts for subschema relations between $(a+b)[S]\mathtt{,}T$ and $a[S]\mathtt{,}T+b[S]\mathtt{,}T$. According to 4.a, the relation may be reduced to the check whether $a[S]\mathtt{,}T$ and $b[S]\mathtt{,}T$ are subschema of $a[S]\mathtt{,}T+b[S]\mathtt{,}T$ (which is trivial in this case). As in [HVP00a], to discuss the item 4.b, let us admit a schema intersection operator $\cap$ such that $S\cap T$ describes the values that belong both to S and T. Then $L[S]\mathtt{,}T$ may be rewritten as $L[S]\mathtt{,Any}\cap\mathtt{Any,}T$ using the fact that `Any` is the greatest schema (see Proposition 2.2.5). For example:

$$\begin{aligned} L_1[S_1]\mathtt{,}T_1+L_2[S_2]\mathtt{,}T_2 =&(L_1[S_1]\mathtt{,Any}\cap\mathtt{Any,}T_1)+(L_2[S_2]\mathtt{,Any}\cap\mathtt{Any,}T_2)\\ =&(L_1[S_1]\mathtt{,Any}+L_2[S_2]\mathtt{,Any})\cap(\mathtt{Any,}T_1+\mathtt{Any,}T_2)\\ &\cap(L_1[S_1]\mathtt{,Any}+\mathtt{Any,}T_2)\cap(\mathtt{Any,}T_1+L_2[S_2]\mathtt{,Any}) \end{aligned}$$

where the last equality follows by distributivity of $\cap$ with respect to union. Therefore, if one intends to derive that $L[S]\mathtt{,}T$ is a subschema of $L_1[S_1]\mathtt{,}T_1+L_2[S_2]\mathtt{,}T_2$ when $\widehat{L}\subseteq\widehat{L_1}\cap\widehat{L_2}$, it is possible to reduce to:

$$\text{for every } J\subseteq\{1,2\} \text{ either}\quad S\ \mathcal{R}\sum_{j\in J}S_j \quad\text{or}\quad T\ \mathcal{R}\sum_{j\in\{1,2\}\setminus J}T_j$$

This is exactly the item 4.b when $I=\{1,2\}$. A particular case is when $I=\{1\}$. For example verifying that $a[S]\mathtt{,}T$ is a subschema of $(a+b)[S']\mathtt{,}T'$. In this case the subsets of I are $\emptyset$ and $\{1\}$ and one is reduced to prove (we let $\sum_{j\in\emptyset}S_j=\mathtt{Empty}$):

$$\Big(S\,\mathcal{R}\,\mathtt{Empty}\quad\text{or}\quad T\,\mathcal{R}\,T'\Big)\quad\text{and}\quad\Big(S\,\mathcal{R}\,S'\quad\text{or}\quad T\,\mathcal{R}\,\mathtt{Empty}\Big)$$

That is, when S and T are not subschema of `Empty`, we are reduced to $S\ \mathcal{R}\ S'$ and $T\ \mathcal{R}\ T'$.

The schemas AnyChan, AnyRecord and Any defined as:

$$\mathcal{E}(\texttt{AnyChan}) = \langle\texttt{Empty}\rangle^{\texttt{O}} + \langle\texttt{Any}\rangle^{\texttt{I}}$$
$$\mathcal{E}(\texttt{AnyRecord}) = \{\ \}$$
$$\mathcal{E}(\texttt{Any}) = (\texttt{\textasciitilde}[\texttt{Any}] + \texttt{AnyChan} + \texttt{AnyRecord})\texttt{,Any} + \texttt{()}$$

own relevant properties. AnyChan collects all the reference schemas, no matter what they can carry; AnyRecord collects all the record schema; Any collects all the documents, namely possibly empty sequences of documents, including channel schemas, no matter how they are labelled. We observe that $\langle\texttt{Empty}\rangle^{\texttt{O}}$ and $\langle\texttt{Any}\rangle^{\texttt{O}}$ are very different. $\langle\texttt{Empty}\rangle^{\texttt{O}}$ collects every reference to remote operations with either capability "O" or "IO", $\langle\texttt{Any}\rangle^{\texttt{O}}$ refers only to operations where arbitrary data can be sent. For instance $\langle a[\ ]\rangle^{\texttt{O}}$ is a subschema of $\langle\texttt{Empty}\rangle^{\texttt{O}}$ but not of $\langle\texttt{Any}\rangle^{\texttt{O}}$. The channel schemas $\langle\texttt{Any}\rangle^{\texttt{I}}$ and $\langle\texttt{Empty}\rangle^{\texttt{I}}$ are different as well. $\langle\texttt{Any}\rangle^{\texttt{I}}$ refers to references that may receive arbitrary data; $\langle\texttt{Empty}\rangle^{\texttt{I}}$ refers to a reference that cannot receive anything.

We also remark about differences between labelled schemas and channel schemas. Let $S = a[\texttt{Blist}] + a[\texttt{Btree}]$ and $S' = a[\texttt{Blist} + \texttt{Btree}]$. Then $S \approx R'$. However $T = \langle\texttt{Blist}\rangle^{\kappa} + \langle\texttt{Btree}\rangle^{\kappa}$ is not subschema-equivalent to $T' = \langle\texttt{Blist} + \texttt{Btree}\rangle^{\kappa}$. Let us discuss the case $\kappa = \texttt{I}$ that is similar to $L[\cdot]$ because covariant. It is possible to prove that $T <: T'$. However the converse is false because references in T may be invoked only with documents that are lists of booleans or only with documents that are trees of booleans. Channels in T' may be invoked with documents belonging either to Blist or to Btree.

Proposition 2.2 *A few properties of $<:$ are in order:*

1. *(Contravariance of $\langle\cdot\rangle^{\texttt{O}}$) $S <: T$ if and only if $\langle T\rangle^{\texttt{O}} <: \langle S\rangle^{\texttt{O}}$;*
2. *(Covariance of $\langle\cdot\rangle^{\texttt{I}}$) $S <: T$ if and only if $\langle S\rangle^{\texttt{I}} <: \langle T\rangle^{\texttt{I}}$;*
3. *(Invariance of $\langle\cdot\rangle^{\texttt{IO}}$) $S <: T$ and $T <: S$ if and only if $\langle S\rangle^{\texttt{IO}} <: \langle T\rangle^{\texttt{IO}}$;*
4. *If S is empty then $S <: \texttt{Empty}$;*
5. *For every S, $\texttt{Empty} <: S <: \texttt{Any}$ and $\langle S\rangle^{\kappa} <: \texttt{AnyChan}$ and $\langle\texttt{Any}\rangle^{\texttt{IO}} <: \langle S\rangle^{\texttt{O}}$ and $\langle\texttt{Empty}\rangle^{\texttt{IO}} <: \langle S\rangle^{\texttt{I}}$ and $\{(\texttt{m}_h{:}S_h)^{h\in H}\} <: \texttt{AnyRecord}$.*

Proof: We demonstrate item 5 (other items are straightforward). Let $\mathcal{R}$ be the least relation containing the identity and the pairs:

$$(\texttt{Empty}, S), (S, \texttt{Any}), (\langle S\rangle^{\kappa}, \texttt{AnyChan}), (\langle\texttt{Any}\rangle^{\texttt{IO}}, \langle S\rangle^{\texttt{O}}), (\langle\texttt{Empty}\rangle^{\texttt{IO}}, \langle S\rangle^{\texttt{I}})$$
$$(\{(\mathtt{m}_h{:}S_h)^{h\in H}\}, \texttt{AnyRecord}), (S, (\texttt{AnyRecord} + \texttt{AnyChan} + \texttt{\~{}[Any]})\texttt{,Any} + \texttt{()})$$

The proof that $\mathcal{R}$ is a subschema relation is straightforward, except for the pairs $(S, \texttt{Any})$ and $(S, (\texttt{AnyRecord} + \texttt{AnyChan} + \texttt{\~{}[Any]})\texttt{,Any} + \texttt{()})$. We analyze the first pair, the other being similar. We show that every R such that $S \downarrow R$ is simulated by $\texttt{Any}$. The interesting case is when $R = L[S']\texttt{,}S''$. In this case $\texttt{Any} \downarrow \texttt{\~{}[Any],}(\texttt{AnyRecord} + \texttt{AnyChan} + \texttt{\~{}[Any]})\texttt{,Any} + \texttt{()}$ and we are in case 4.b of Definition 2.2. Since $S \downarrow R$ then S is not-empty, similarly for $\texttt{Any}$. Therefore we are reduced to $(S', \texttt{Any}), (S'', (\texttt{AnyRecord} + \texttt{AnyChan} + \texttt{\~{}[Any]})\texttt{,Any} + \texttt{()}) \in \mathcal{R}$, which hold by definition. □

Lemma 2.1 $<:$ *is reflexive and transitive.*

Proof: Reflexivity is straightforward. As regards transitivity, let $\mathcal{R}^+$ be the least relation that contains $\mathcal{R}$ and closed under the following operations:

1. if $S\,\mathcal{R}^+\,T$ then $S\,\mathcal{R}^+\,T + R$;
2. if $S\,\mathcal{R}^+\,T$ and $S'\,\mathcal{R}^+\,T$ then $S + S'\,\mathcal{R}^+\,T$;
3. if $S\,\mathcal{R}^+\,T$ and $S \downarrow L[S']\texttt{,}S''$ then $L'[S']\texttt{,}S''\,\mathcal{R}^+\,T$ with $\widehat{L'} \subseteq \widehat{L}$;

It is not difficult to verify that, if $\mathcal{R}$ is a subschema relation, then $\mathcal{R}^+$ is a subschema, as well.

Let $\mathcal{R}$ and $\mathcal{S}$ be two subschema relations such that $S\,\mathcal{R}\,T$ and $T\,\mathcal{S}\,R$. We prove that

$$\mathcal{T} = \{(S, R) \quad | \quad S\,\mathcal{R}^+\,T \text{ and } T\mathcal{S}^+R\}$$

is a subschema relation. Let $S\ \mathcal{T}\ R$. The critical case is when $S \downarrow L[S']\texttt{,}S''$. According to the definition of $\mathcal{T}$, there exists T such that $S\,\mathcal{R}^+\,T$ and $T\mathcal{S}^+R$. By Definition 2.2, $T \downarrow L'[T']\texttt{,}T''$ with $\widehat{L} \cap \widehat{L'} \neq \emptyset$. There are two cases:

(a) $T \downarrow L'[T']\texttt{,}T''$ with $\widehat{L} \not\subseteq \widehat{L'}$ and $\widehat{L} \cap \widehat{L'} \neq \emptyset$. We are reduced to $(L \cap L')[S']\texttt{,}S''\ \mathcal{T}\ R$ and $(L \setminus L')[S']\texttt{,}S''\ \mathcal{T}\ R$, which are immediate by definition of $\mathcal{T}$.

(b) $T \downarrow L_i[T'_i], T''_i$ with $i \in I$ and $\widehat{L} \subseteq \bigcap_{i\in I} \widehat{L_i}$ and, for every $K \subseteq I$:

$$\text{either} \quad S' \mathcal{R} \sum_{k\in K} T'_k \quad \text{or} \quad S'' \mathcal{R} \sum_{k\in I\setminus K} T''_k \,. \tag{2.1}$$

There are two subcases:

(b1) $R \downarrow M[R'], R''$ with $\widehat{L} \cap \widehat{M} \neq \emptyset$ and $\widehat{L} \not\subseteq \widehat{M}$. In this case we must prove $(L \cap M)[S'], S'' \;\mathcal{T}\; R$ and $(L \setminus M)[S'], S'' \;\mathcal{T}\; R$, which are immediate by definition of $\mathcal{T}$.

(b2) $R \downarrow M_j[R'_j], R''_j$ with $j \in J$ and $\widehat{L} \subseteq \bigcap_{j\in J} \widehat{M_j}$. There are again two subcases: (b2.1) there are i, k such that $\widehat{L_i} \not\subseteq \widehat{M_k}$; (b2.2) the contrary of (b2.1). In case (b2.1) we apply the simulation case 4.(a): it must be $(L_i \cap M_k)[T'_i], T''_i \;\mathcal{S}\; R$ and $(L_i \setminus M_k)[T'_i], T''_i \;\mathcal{S}\; R$. As far as $(L_i \cap M_k)[T'_i], T''_i \;\mathcal{S}\; R$ is concerned, there are two remarks: (i) $\widehat{L} \subseteq \widehat{L_i} \cap \widehat{M_k}$ and (ii) there exists a set J_i such that $R \downarrow M_j[R'_j], R''_j$ with $j \in J_i$ and $\widehat{L_i} \subseteq \bigcap_{j\in J_i} \widehat{M_j}$ and, for every $K' \subseteq J_i$:

$$\text{either} \quad T'_i \,\mathcal{S} \sum_{k\in K'} R'_k \quad \text{or} \quad T''_i \,\mathcal{S} \sum_{k\in J_i\setminus K'} R''_k \tag{2.2}$$

Let J contain all the J_i that correspond to case (b2.1). Let $K \subseteq J$. Since $\widehat{L} \subseteq \bigcap_{j\in J} \widehat{M_j}$, we must prove:

$$\text{either} \quad S' \;\mathcal{T} \sum_{k\in K} R'_k \qquad \text{or} \qquad S'' \;\mathcal{T} \sum_{k\in J\setminus K} R''_k \tag{2.3}$$

For every $i \in I$, the constraint (2.2) implies

$$\text{either} \quad T'_i \;\mathcal{S}^+ \sum_{k\in J} R'_k \quad \text{or} \quad T''_i \;\mathcal{S}^+ \sum_{k\in J\setminus K} R''_j \tag{2.4}$$

where the relation is $\mathcal{S}^+$. Let $H_K = \{h \in I \mid T'_h \;\mathcal{S}^+ \; \sum_{k\in K} R'_k\}$. By definition $H_K \subseteq I$ and $T''_{h'} \;\mathcal{S}^+ \; \sum_{k\in J\setminus K} R''_k$ for every $h' \in I \setminus H_K$. The constraint (2.1) implies

$$\text{either} \quad S' \;\mathcal{R}^+ \sum_{h\in H_K} T'_h \quad \text{or} \quad S'' \;\mathcal{R}^+ \sum_{h\in I\setminus H_K} T''_h \tag{2.5}$$

The constraint (2.3) follows from (2.5) and (2.4).

The case (b2.2) is similar to (b2.1) but we apply the simulation case 4.(b) $\square$

Lemma 2.2 $S\mathbin{,}(S_1+S_2) <: S\mathbin{,}S_1 + S\mathbin{,}S_2$.

Proof: Let

$$\mathcal{R} = \{(T\mathbin{,}(T_1+T_2),\ T\mathbin{,}T_1+T\mathbin{,}T_2),\ (T,T) :\ T \text{ and } T_1 \text{ and } T_2 \text{ are schemas}\}.$$

We verify that $\mathcal{R}$ is a subschema relation. We only need to verify pairs of the shape $(T\mathbin{,}(T_1+T_2), T\mathbin{,}T_1+T\mathbin{,}T_2)$. We distinguish two cases.

- If $T \downarrow \texttt{()}$, by definition of handle if $T\mathbin{,}(T_1+T_2) \downarrow R$ with either $T_1 \downarrow R$ or $T_2 \downarrow R$ then $T\mathbin{,}T_1+T\mathbin{,}T_2 \downarrow R$. Tince $(R,R) \in \mathcal{R}$, we conclude.
- If $T \downarrow R\mathbin{,}T'$ – with R either $R = \langle R' \rangle^\kappa$ or $R' = \{(\mathtt{m}_h{:}S_h)^{h\in H}\}$ or $R = L[R']$ – by definition of handle $T\mathbin{,}(T_1+T_2) \downarrow R\mathbin{,}T'\mathbin{,}(T_1+T_2)$, and $T\mathbin{,}T_1 \downarrow R\mathbin{,}T'\mathbin{,}T_1$, and $T\mathbin{,}T_2 \downarrow R\mathbin{,}T'\mathbin{,}T_2$. Then by Definition of $<:$, item 2, item 3, or item 4.b we have to verify that $(T'\mathbin{,}(T_1+T_2),\ T'\mathbin{,}T_1+T'\mathbin{,}T_2) \in \mathcal{R}$. This holds by definition of $\mathcal{R}$. □

2.5 Primitive types

The extension of our schema language with primitive types is not difficult. Consider the new syntax:

B ::=		**primitive types**	S ::=		**schema**
	`n`	(integer constant)		$\cdots$	
	`s`	(string constant)		`B`	(primitive types)
	`Int`	(integers)			
	`String`	(strings)			

The primitive types `n`, `s`, `Int`, and `String` respectively describe a specific integer, a specific string, the set of integers, and the set of strings. For example, the schema that collects integers and strings is `Int` + `String`; the schema that collects references with integer messages is $\langle\texttt{Int}\rangle^{\texttt{i}} + \langle\texttt{Int}\rangle^{\texttt{o}}$. As regards the subschema relation, the handles are extended with $\texttt{B} \downarrow \texttt{B,()}$. Let $\leq_b$ be the least partial order on primitive types such that $\texttt{n} \leq_p \texttt{Int}$ and $\texttt{s} \leq_p \texttt{String}$. To define the subschema relation for the new language it sufficies to extend Definition 2.2 with

5. *$S \downarrow$ B,S' implies $T \downarrow$ B$'_i$,T'_i, for $1 \leq i \leq n$, with B $\sqsubseteq$ B$'_i$ and $S' \mathcal{R} \sum_{1\leq i\leq n} T'_i$;*

Then a set of handles B$_i$,T'_i of the greater schema is selected such that B is smaller than B$_i$ and S' is smaller than the union of the T'_i's. It follows that $1 + \texttt{Int} <: \texttt{Int}$, and 5,$(S + S') <: \texttt{Int},S + 5,S'$, and $a[1 + \texttt{"bye"}] <: a[1] + a[\texttt{"bye"}]$ (the proofs are left to the reader).

Remark. The algorithm for computing the subschema relation is similar to the one developed for XDuce [HVP00a]. It is computationally expensive: the cost of the algorithm for subschema is exponential in the sizes of the schemas.

2.6 The algorithmic subschema

Table 2.2 illustrates the algorithmic version of <:. In this case the subschema relationship between S and T is demonstrated by a proof tree with a conclusion $S \preceq_{\mathtt{A}} T \Rightarrow \mathtt{A}'$, where $\mathtt{A} = \emptyset$. In this proof tree A is filled with pairs (S,T) (called assumptions) meaning that the subschema relation between S and T has been proved or is being proved. Since S and T are regular (i.e. finite state) this ensures the termination of the algorithm (see Section 2.6.1 for the formal proof). Following Brand and Henglein [BH97], the resulting set of pairs A$'$ is used for building the remaining proof tree. We remark that the algorithm may be simplified omitting A$'$ (which is tedious to keep in the formalization). However this simplification yields an less efficient procedure because it does not remember pairs of schemas across the recursive calls [AC93].

Rules (void), (chan), (split), and (lseq) correspond to items 1, 2.1, 2.2, 2.3, 4.1, and 4.2 of <:. In order to have a more readable rule, item 3 of <: is implemented by two rules: (record) and (record-aux) where schemas are records. (union) distributes the union schema on top of a sequence verifying that both the components S,S'' and S',S'' are subschema of T. (nameh) expands the constant schema name to its definition adding a new assumption in A. (namel) closes the proof tree when a prove have been already done. (empty) takes into account the case when S does not have any handle returning success. This is necessary, for instance, for proving $\mathtt{U} \preceq_{\emptyset} b[\ ]$ with $\mathcal{E}(\mathtt{U}) = a[\mathtt{U}]$ where – after an application of (nameh) – (lseq) fails because of the false premise $\widehat{a} \subseteq \widehat{b}$. We note that, strictly speaking, the algorithm is not syntax directed because of this last rule that

Table 2.2: The algorithmic subschema (arguments of shape $\langle S\rangle^\kappa$ are always replaced by $\langle S\rangle^\kappa\texttt{,()}$. Similarly for $L[S]$, $\mathtt{U}$, S^*, $\{(\mathtt{m}_h{:}S_h)^{h\in H}\}$ and $S+S'$. Arguments $\texttt{()},S$ are always replaced by S).

(VOID)
$$\frac{T\downarrow \texttt{()}}{\texttt{()}\preceq_{\mathtt{A}} T\Rightarrow \mathtt{A}}$$

(EMPTY)
$$\frac{\mathsf{handles}(S)=\emptyset}{S\preceq_{\mathtt{A}} T\Rightarrow \mathtt{A}}$$

(CHAN)
$$\frac{\begin{array}{c}(T\downarrow \langle T_i\rangle^{\kappa_i}\texttt{,}T'_i)^{i\in 1..n}\quad \kappa\le\kappa_i\\ \left(\begin{array}{r}\kappa_i=\mathtt{O}\quad\text{implies}\quad T_i\preceq_{\mathtt{A}_{i-1}} S\Rightarrow \mathtt{A}_i\\ \kappa_i=\mathtt{I}\quad\text{implies}\quad S\preceq_{\mathtt{A}_{i-1}} T_i\Rightarrow \mathtt{A}_i\\ \kappa_i=\mathtt{IO}\quad\text{implies}\quad S\preceq_{\mathtt{A}_{i-1}} T_i\Rightarrow \mathtt{A}'_i \text{ and } T_i\preceq_{\mathtt{A}'_i} S\Rightarrow \mathtt{A}_i\end{array}\right)^{i\in 1..n}\\ S'\preceq_{\mathtt{A}_n}\sum_{i\in 1..n} T'_i\Rightarrow \mathtt{A}_{n+1}\end{array}}{\langle S\rangle^\kappa\texttt{,}S'\preceq_{\mathtt{A}_0} T\Rightarrow \mathtt{A}_{n+1}}$$

(RECORD)
$$\frac{\begin{array}{c}T\downarrow(\{(\mathtt{m}_j{:}T_j)^{j\in J_i}\}\texttt{,}T'_i)^{i\in\{1,\dots,n\}}\\ (\{(\mathtt{m}_h{:}S_h)^{h\in H}\}\preceq_{\mathtt{A}_{h_{i-1}}}\{(\mathtt{m}_j{:}T_j)^{j\in J_i}\}\Rightarrow \mathtt{A}_{h_i})^{i\in\{1,\dots,n\}}\quad S'\preceq_{\mathtt{A}_{h_n}}\sum_{i\in\{1,\dots,n\}} T'_i\Rightarrow \mathtt{A}_{h_{n+1}}\end{array}}{\{(\mathtt{m}_h{:}S_h)^{h\in H}\}\texttt{,}S'\preceq_{\mathtt{A}_{h_0}} T\Rightarrow \mathtt{A}_{h_{n+1}}}$$

(RECORD-AUX)
$$\frac{\{h_1,\dots,h_n\}=J\subseteq H\quad (S_{h_{i-1}}\preceq_{\mathtt{A}_{h_{i-1}}} T_{h_i})^{i\in\{1,\dots,n\}}}{\{(\mathtt{m}_h{:}S_h)^{h\in H}\}\texttt{,()}\preceq_{\mathtt{A}_{h_0}}\{(\mathtt{m}_j{:}T_j)^{j\in J}\}\texttt{,()}\Rightarrow \mathtt{A}_{h_n}}$$

(SPLIT)
$$\frac{\begin{array}{c}T\downarrow L'[T']\texttt{,}T''\quad \widehat{L}\not\subseteq\widehat{L'}\quad \widehat{L}\cap\widehat{L'}\neq\emptyset\\ (L\setminus L')[S]\texttt{,}S'\preceq_{\mathtt{A}} T\Rightarrow \mathtt{A}'\quad (L\cap L')[S]\texttt{,}S'\preceq_{\mathtt{A}'} T\Rightarrow \mathtt{A}''\end{array}}{L[S]\texttt{,}S'\preceq_{\mathtt{A}} T\Rightarrow \mathtt{A}''}$$

(LSEQ)
$$\frac{\begin{array}{c}(T\downarrow L_i[T_i]\texttt{,}T'_i)^{i\in\{1,\dots,n\}}\quad \widehat{L}\subseteq\bigcap_{i\in\{1,\dots,n\}}\widehat{L}_i\{J_1,\dots,J_m\}=\mathcal{P}(\{1,\dots,n\})\\ \left(S\preceq_{\mathtt{A}_{k-1}}\sum_{i\in J_k} T_i\Rightarrow \mathtt{A}_k\quad\mathtt{or}\quad S'\preceq_{\mathtt{A}_{k-1}}\sum_{i\in\{1,\dots,n\}\setminus J_k} T'_i\Rightarrow \mathtt{A}_k\right)^{k\in 1..m}\end{array}}{L[S]\texttt{,}S'\preceq_{\mathtt{A}_0} T\Rightarrow \mathtt{A}_m}$$

(UNION)
$$\frac{S\texttt{,}S''\preceq_{\mathtt{A}} T\Rightarrow \mathtt{A}'\quad S'\texttt{,}S''\preceq_{\mathtt{A}'} T\Rightarrow \mathtt{A}''}{(S+S')\texttt{,}S''\preceq_{\mathtt{A}} T\Rightarrow \mathtt{A}''}$$

(NAMEH)
$$\frac{\mathtt{A}'=\mathtt{A}\cup\{(\mathtt{U},S,T)\}\quad \mathcal{E}(\mathtt{U})\texttt{,}S\preceq_{\mathtt{A}'} T\Rightarrow \mathtt{A}''}{\mathtt{U}\texttt{,}S\preceq_{\mathtt{A}} T\Rightarrow \mathtt{A}''}$$

(NAMEL)
$$\frac{(S,T)\in \mathtt{A}}{S\preceq_{\mathtt{A}} T\Rightarrow \mathtt{A}}$$

can be always applied and because of the two rules for records. An implementation would first verify the schema to have some handles proceeding with one of the syntax-directed rules applying (RECORD-AUX) instead of (RECORD) when it is possible.

2.6.1 Properties of the algorithmic subschema

We note that the rules in Table 2.2 defines a program, called the $\preceq$*-program*, that takes a triple $(S, T, \mathtt{A})$ and attempts to build the proof tree by recursively analyzing the structure of S and the set $\mathtt{A}$. The program returns a set $\mathtt{A}'$ if the attempt succeeds, returns a failure otherwise. In this section we prove basic properties of $S \preceq_{\mathtt{A}} T \Rightarrow \mathtt{A}'$ such as soundness, termination and completeness.

Soundness The relation $\preceq$ is sound with respect to $<:$. To demonstrate this property we consider the subschema relation made by pairs in the derivation tree of $S \preceq_{\emptyset} T \Rightarrow \mathtt{A}$.

Lemma 2.3 (Soundness) *If* $S \preceq_{\emptyset} T \Rightarrow \mathtt{A}$ *then* $S <: T$.

Proof: Let $\mathcal{R}$ be the relation containing

1. pairs (S', T') such that a subtree $S' \preceq_{\mathtt{A}'} T' \Rightarrow \mathtt{A}''$ exists in the tree $S \preceq_{\emptyset} T \Rightarrow \mathtt{A}$;
2. if $(\mathtt{B}, T') \in \mathcal{R}$ then $(\mathtt{B}\,\mathtt{,}\,\mathtt{()}\,\mathtt{,}\,T') \in \mathcal{R}$, too. Similarly for pairs $(\langle S' \rangle^{\kappa}, T')$, $(\langle S' \rangle^{\kappa}, T')$, $(L[S'], T')$, $(\{(\mathtt{m}_h{:}S_h)^{h \in H}\}, T')$, $(S' + S'', T')$, and $(\mathtt{U}, T')$.

To check that $\mathcal{R}$ is a subschema relation, let $(S', T') \in \mathcal{R}$ and $S' \downarrow R$. By induction on the structure of the proof $S' \downarrow R$ it is easy to show that $(R, T) \in \mathcal{R}$, too. $\square$

Termination The $\preceq$-program terminates. To demonstrate this property we start with some prelimars. Let $\mathtt{subt}(S)$, called the *set of subterms* of S, be the smallest set satisfying the following equations:

$$
\begin{array}{rcl}
\mathtt{t}(\mathtt{()}) & = & \{\mathtt{()}\} \\
\mathtt{t}(\mathtt{U}) & = & \{\mathtt{U}\} \cup \{\mathtt{U}\,\mathtt{,}\,\mathtt{()}\} \cup \{\mathtt{t}(\mathcal{E}(\mathtt{U}))\} \\
\mathtt{t}(\langle S \rangle^{\kappa}) & = & \{\langle S \rangle^{\kappa}\} \cup \{\langle S \rangle^{\kappa}\,\mathtt{,}\,\mathtt{()}\} \cup \mathtt{t}(S) \\
\mathtt{t}(L[S]) & = & \{L[S]\} \cup \{L[S]\,\mathtt{,}\,\mathtt{()}\} \cup \mathtt{subt}(S) \\
\mathtt{t}(\{(\mathtt{m}_h{:}S_h)^{h \in H}\}) & = & \{\{(\mathtt{m}_h{:}S_h)^{h \in H}\}\} \cup \{\{(\mathtt{m}_h{:}S_h)^{h \in H}\}\,\mathtt{,}\,\mathtt{()}\} \cup \bigcup_{h \in H} \mathtt{t}(S_h) \\
\mathtt{t}(S\,\mathtt{,}\,S') & = & \{T\,\mathtt{,}\,S' \mid T \in \mathtt{subt}(S)\} \\
\mathtt{t}(S + T) & = & \{(S + T)\,\mathtt{,}\,\mathtt{()}\} \cup \mathtt{t}(S) \cup \mathtt{t}(T)
\end{array}
$$

Let also $\texttt{anames}(S) \stackrel{\text{def}}{=} \{\texttt{U},T : \texttt{U},T \in \mathfrak{t}(S)\}$ and $\texttt{lsubt}(S,T)$ be the smallest set containing $\mathfrak{t}(S)$, $\mathfrak{t}(T)$ and closed under the following properties:

- if $L[Q]\texttt{,}Q' \in \texttt{lsubt}(S,T)$ and $L'[Q'']\texttt{,}Q''' \in \texttt{lsubt}(S,T)$ and $\widehat{L} \not\subseteq \widehat{L'}$ then $(L \setminus L')[Q]\texttt{,}Q' \in \texttt{lsubt}(S,T)$ and $(L \cap L')[Q]\texttt{,}Q' \in \texttt{lsubt}(S,T)$;

- if $S\texttt{,}S' \in \mathfrak{t}(S)$ and $T\texttt{,}T' \in \mathfrak{t}(S)$ then $S' + T' \in \mathfrak{t}(S)$.

It is easy to see that $\mathfrak{t}(S)$ is finite and, consequentially, $\texttt{lsubt}(S,T)$ is also finite. Finally, let $\|S\|_\emptyset$, called the *size* of S, be the function inductively defined as:

$$\|S\|_{\mathcal{X}} = \begin{cases} 0 & S = \texttt{U} \in \mathcal{X} \\ 1 & S = \texttt{()} \\ \|\mathcal{E}(\texttt{U})\|_{\mathcal{X}\cup\{\texttt{U}\}} & S = \texttt{U} \notin \mathcal{X} \\ 1 + \|T\|_{\mathcal{X}} & S = \langle T \rangle^{\kappa} \text{ or } S = L[T] \\ 1 + \|T\|_{\mathcal{X}} + \|T'\|_{\mathcal{X}} & S = T\texttt{,}T' \text{ or } S = T + T' \\ 1 + \sum_{h\in H} \|T_h\| & S = \{(\texttt{m}_h : T_h)^{h\in H}\} \end{cases}$$

In the following $\|S\|_\emptyset$ will be shortened into $\|S\|$. We note that $\|S\|$ and $\mid \mathfrak{t}(S) \mid$ are different. For instance $\|S + S\| = 2 \times \|S\| + 1$ whilst $\mid \mathfrak{t}(S + S) \mid = \mid \mathfrak{t}(S) \mid +1$.

Lemma 2.4 (Termination) *The $\preceq$-program terminates.*

Proof: Let $n_{S,T,\texttt{A}} = \mid (\texttt{anames}(S) \cup \texttt{anames}(T)) \times \texttt{lsubt}(S,T) \setminus \texttt{A} \mid$ (the subtrees of T are considered because of the contravariance of $\langle\cdot\rangle^{\texttt{O}}$). We note that $\texttt{A}$ is contained into $(\texttt{anames}(S) \cup \texttt{anames}(T)) \times \texttt{lsubt}(S,T)$. We demonstrate that every invocation of $S \preceq_{\texttt{A}} T \Rightarrow \texttt{A}'$ in the premises of the rules of Table 2.2 decreases the value $(n_{S,T,\texttt{A}}, \|S\|+\|T\|)$ (the order is lexicographic) of the conclusion. In particular (VOID), (EMPTY) and (NAMEL) do not generate any new judgment. (NAMEH) increases the cardinality of the set of assumptions thus decreasing the structure of the pair. In case of (CHAN), (LSEQ), (UNION) and (RECORD-AUX) $\|S\| + \|T\|$ diminish leaving unchanged the set of assumption. There are two problematic cases: when the $\preceq$-program trys to apply (SPLIT) and (RECORD). In this case, the value $(n_{S,T,\texttt{A}}, \|S\| + \|T\|)$ for the two premises is equal to that of the conclusion. However, after a finite number of application of (SPLIT) – corresponding (in worst case) to the number of labelled handles of T, which are finite by (1) – (SPLIT) reduces to (LSEQ). In (LSEQ) the value $(n_{S,T,\texttt{A}}, \|S\| + \|T\|)$ decreases, thus guaranteeing termination.

In case of (RECORD), at most in two steps requiring (RECORD-AUX), $\|S\| + \|T\|$ diminish and we conclude. This says that the algorithm terminates. □

Completeness The relation $\preceq$ is complete with respect to $<:$. To demonstrate this basic property of the $\preceq$-program we show that the algorithm cannot fail when it is invoked on schemas related by $<:$. This, together with the termination property guarantees completeness.

Definition 2.3 *We say that a judgment $S \preceq_{\mathtt{A}} T \Rightarrow \mathtt{A}'$ is* correct *if and only if: (1) $S <: T$ and (2) $S <: T$ for every $(S, T) \in \mathtt{A}$.*

Proposition 2.3 *If $S \preceq_{\mathtt{A}} T \Rightarrow \mathtt{A}'$ is correct then $S' <: T'$ for every $(S', T') \in \mathtt{A}'$.*

Proof: We reason by induction on the derivation of $S \preceq_{\mathtt{A}} T \Rightarrow \mathtt{A}'$. The only interesting rule is (NAMEH) because it augments with the pair $(\mathtt{U}, S, T)$ the set of assumptions. $\mathtt{U}, S <: T$ follows immediately by the correctness of the current judgment. □

Proposition 2.4 *If $S \preceq_{\mathtt{A}} T \Rightarrow \mathtt{A}'$ is correct then one of rule in Table 2.2 applies and the judgments corresponding to the premise are correct.*

Proof: If $\mathsf{handle}(S) = \emptyset$ then (EMPTY) applies and no judgments are generated. Otherwise, we distinguish several cases depending on the structure of S. If $S = \mathtt{()}$ then (VOID) applies generating no judgements. If $S = \mathtt{B}, S'$ then (BASE) applies generating the judgement $S' \preceq_{\mathtt{A}} \sum_{i\in\{1,\ldots,n\}} T_i \Rightarrow \mathtt{A}'$ that is correct by the hypotheses on $\mathtt{A}$ and by $S <: \sum_{i\in\{1,\ldots,n\}} T_i$. If either $S = \langle S' \rangle^{\kappa}, S''$ or $S = L[S'], S''$ or $S = \{(\mathtt{m}_h{:}S_h)^{h\in H}\}$ then either (CHAN) or (SPLIT) or (LSEQ) or (RECORD) or (RECORD-AUX) can by applied. The correctness of the new judgements follows by the hypothesis $S <: T$ and by Proposition 2.3.

If $S = (S' + S''), S'''$ then the rule (UNION) applies and we obtain two judgements: *a*) $S', S''' \preceq_{\mathtt{A}} T \Rightarrow \mathtt{A}'$, and *b*) $S'', S''' \preceq_{\mathtt{A}'} T \Rightarrow \mathtt{A}''$. By the hypothesis $S <: T$ and by definition of handle we obtain $S', S''' <: T$ and $S'', S''' <: T$. Hence, by Proposition 2.3, we conclude that *a*) and *b*) are correct.

If $S = \mathtt{U}, S'$ and $(S, T) \notin \mathtt{A}$ then either (NAMEH) or (NAMEL) can be applied. In case of (NAMEH), the correctness of the new judgement follows by the correctness of the current judgement. In case of (NAMEL), no new judgement is generated. □

Lemma 2.5 (Completeness) *If* $S <: T$ *then* $S \preceq_{\emptyset} T$.

Proof: We consider the derivation produced by the algorithm $S \preceq_{\emptyset} T$. Since $S \preceq_{\emptyset} T$ is correct, by Proposition 2.4, only correct judgments are created and the algorithm can only proceed or return success. Therefore the algorithm cannot fail when started with a correct input. Since, by Lemma 2.4, the algorithm is terminating, we conclude that it succeeds. □

2.7 Label-determined schemas

It is clear that rule 4.b of $<:$ – implemented by (LSEQ) – yields an exponential algorithm because it requires a universal quantification over the set of subset of handles. Indeed – if we omit references – the relation $<:$ computes tree language containement, that turns out to be computationally expensive – it has an exponential cost with respect to the sizes of the schemas [CDG+97]. This is an issue in Web-services, where data coming from untrusted parties, such as WSDL documents (containing the schema of a service), might be validated at run-time before processing. While validation has a polynomial cost with respect to the size of the datum in current schema languages, this is not so when data carry references. In these cases, validation has to verify that the schema of the reference conforms with some expected schema, thus reducing itself to the subschema relation. It is worth to notice that in XDUCE run-time subschema checks are avoided because programs are strictly coupled and typechecking guarantees that invalid values cannot be produced. In CDUCE there is the possibility of using pattern matching on function value that requiring the subschema relation at run-time, but this feature is never used in CDUCE programs [Cas06].

To avoid significative run-time degradations of Web-services technologies, we impose a language restriction to diminish the cost of the subschema relation. The restriction prevents unions of schemas having a common starting tag and is similar to the restriction used in single-type tree grammars [MLM01]. Specifically, similarly to XML-SCHEMA, we constrain schemas to retain a deterministic model as regards tag-labeled transitions. The model is still nondeterministic with respect to other transitions. For the resulting schemas, called *label-determined*, the subschema relation can be implemented in polynomial time with respect to the size of the schemas. We proceed by showing a simplification of the subschema relation – the determined subschema relation – and then we show that it is

equivalent, for determined schemas to <:. Then we demonstrate that it has a polynomial cost with respect to the sizes of the schemas. This result extends to reference schemas the computational complexity of language difference for deterministic tree automata (and XML schemas) computed in [CDG+97].

Definition 2.4 *The set* ldet *of* label-determined schemas *is the greatest set of schemas such that:*

$\texttt{()} \in \mathsf{ldet}$	
$\langle S\rangle^\kappa \in \mathsf{ldet}$	if $S \in \mathsf{ldet}$
$\{(\mathtt{m}_h{:}S_h)^{h\in H}\} \in \mathsf{ldet}$	if, for every h, $S_h \in \mathsf{ldet}$
$L[S] \in \mathsf{ldet}$	if $S \in \mathsf{ldet}$
$S\,\texttt{,}T \in \mathsf{ldet}$	if $S \in \mathsf{ldet}$ and $T \in \mathsf{ldet}$
$S + T \in \mathsf{ldet}$	if $S \downarrow L[S']\texttt{,}S''$ and $T \downarrow L'[T']\texttt{,}T''$ implies $\widehat{L} \cap \widehat{L'} = \emptyset$ and $S \in \mathsf{ldet}, T \in \mathsf{ldet}$
$\mathtt{U} \in \mathsf{ldet}$	if $\mathcal{E}(\mathtt{U}) \in \mathsf{ldet}$

By the definition $a[S] + (\texttt{\~{}} \setminus a)[T]$ and $\texttt{\~{}}[S] + \langle S\rangle^\kappa + \langle T\rangle^{\kappa'}$ are label-determined schemas whilst $a[\,] + (a+b)[\,]$ and $\langle a[\,] + \texttt{\~{}}[\,]\rangle^\kappa$ are not label-determined. It is worth to emphatize that every empty schema – the schema that does not retain any handle – is in ldet and that schemas like $a[\,] + a[\texttt{Empty}]$ are also label-determined.

A determined subschema relation is a simplification of the item 4 of the subschema relation <:. Other items are left unchanged. We show that the determined subschema relation is equivalent to <: for label-determined schemas and that it can be implemented by a polynomial algorithm.

Definition 2.5 *A* determined subschema $\mathcal{D}$ *is a relation on schemas such that* $S\mathcal{D}T$ *implies:*

1. $S \downarrow \texttt{()}$ *implies* $T \downarrow \texttt{()}$*;*

2. $S \downarrow \langle S'\rangle^\kappa\texttt{,}S''$ *implies* $T \downarrow \langle T_i\rangle^{\kappa_i}\texttt{,}T_i'$*, for* $1 \le i \le n$*, with* $\kappa \le \kappa_i$*,* $S''\mathcal{D}\sum_{1\le i\le n} T_i'$*, and, for every* $1 \le i \le n$*, one of the following conditions holds:*

   (a) either $\kappa_i = \mathtt{O}$ *and* $T_i'\ \mathcal{R}^-\ S'$*,*

   (b) or $\kappa_i = \mathtt{I}$ *and* $S'\mathcal{D}T_i'$*,*

(c) *or* $\kappa_i = \mathtt{IO}$ *and* $S'\mathcal{D}T_i'$ *and* $T_i'\mathcal{D}S'$;

3. $S \downarrow \{(\mathtt{m}_h{:}S_h)^{h\in H}\}\mathbf{,}S'$ *implies* $T \downarrow \{(\mathtt{m}_{j_i}{:}T_{j_i})^{j_i\in J_i}\}\mathbf{,}T_i'$, *for* $1 \le i \le n$, *with* $S'\mathcal{D}\sum_{1\le i\le n} T_i'$, *and, for every* $1 \le i \le n$ $J_i \subseteq H$, *and,* $S_{j_i}\mathcal{D}T_{j_i}$ *for every* $j_i \in J_i$;

4. $S \downarrow L[S']\mathbf{,}S''$ *implies* $T \downarrow L_i[T_i']\mathbf{,}T_i''$, *for* $1 \le i \le n$ *with* $\widehat{L}\cap\widehat{L_i} \neq \emptyset$, $\widehat{L} \subseteq \bigcup_{i\in\{1,\ldots,n\}} \widehat{L_i}$, $S'\mathcal{D}T_i'$, *and* $S''\mathcal{D}T_i''$.

Let $<:_{\mathtt{d}}$ *be the largest* determined subschema relation.

Proposition 2.5 *If* $S <:_{\mathtt{d}} T$ *then* $S <: T$.

Proof: Let $\mathcal{D}$ a determined subschema relation and let $\mathcal{D}'$ be the least relation containing $\mathcal{D}$ and closed under the following operations:

1. $(L[S]\mathbf{,}S', T) \in \mathcal{D}'$ and $T \downarrow L'[T']\mathbf{,}T''$ implies

$$\big((L \setminus L')[S]\mathbf{,}S', T\big) \in \mathcal{D}' \qquad \text{and} \qquad \big((L \cap L')[S]\mathbf{,}S', T\big) \in \mathcal{D}'$$

2. $(S, T) \in \mathcal{D}'$ implies $(S, T + T') \in \mathcal{D}'$.

We prove that $\mathcal{D}'$ is a subschema relation. Since items 1, 2, 3 are the same, we reduce to prove that condition 4 of $\mathcal{D}$ implies item 4 of $\mathcal{R}$. Let $(S, T) \in \mathcal{D}'$ with $S \downarrow L[S']\mathbf{,}S''$ and $T \downarrow L_i[T_i']\mathbf{,}T_i''$. We distinguish two cases:

1. if there exists $T \downarrow L'[T']\mathbf{,}T''$ with $\widehat{L} \cap \widehat{L'} \neq \emptyset$ and $\widehat{L} \not\subseteq \widehat{L'}$ we need to verify that

$$\big((L \setminus L')[S']\mathbf{,}S'', T\big) \in \mathcal{D}' \qquad \big((L \cap L')[S']\mathbf{,}S'', T\big) \in \mathcal{D}'$$

   which follows by definition of $\mathcal{D}'$ (closure 1).

2. if $\widehat{L} \subseteq \bigcap_{i\in\{1,\ldots,n\}} \widehat{L_i}$ we need to verify that for every $J \subseteq \{1, \ldots, n\}$

$$(S', \sum_{j\in J} T_j') \in \mathcal{D}' \qquad \text{or} \qquad (S'', \sum_{j\in\{1,\ldots,n\}\setminus J} T_j'') \in \mathcal{D}'$$

   Since $(S', T_i') \in \mathcal{D}'$ and $(S'', T_i'') \in \mathcal{D}'$ we conclude by definition of $\mathcal{D}'$ (closure 2). □

Lemma 2.6 *Let* $S \in \mathsf{ldet}$ *and* $T \in \mathsf{ldet}$. $S <: T$ *if and only if* $S <:_{\mathtt{d}} T$.

Proof:

($\Rightarrow$) Follows immediately by Proposition 2.5.

($\Leftarrow$) By the hypothesis $S \in \mathsf{ldet}$ and $T \in \mathsf{ldet}$, item 4.(b) of Definition 2.2 <: applies with $n = 1$ thus reducing item 4 to:

(4) $S \downarrow L[S']\,\mathbf{,}S''$ *then one of the following conditions holds:*

(a) *either* $T \downarrow L'[T']\,\mathbf{,}T''$ *with* $\widehat{L} \cap \widehat{L'} \neq \emptyset$, $\widehat{L} \not\subseteq \widehat{L'}$, $(L \setminus L')[S']\,\mathbf{,}S''\ \mathcal{R}\ T$, *and* $(L \cap L')[S']\,\mathbf{,}S''\ \mathcal{R}\ T$;

(b) *or* $T \downarrow L'[T']\,\mathbf{,}T''$ *with* $\widehat{L} \subseteq \widehat{L'}$ *and* $S'\ \mathcal{R}\ T'$ *and* $S''\ \mathcal{R}\ T''$.

We now show that 4.(a) and 4.(b) are equivalent to condition 4 of $\mathcal{D}$. 4.(a) reduces the subschema relation to verify $L'_1[S']\,\mathbf{,}S''\ \mathcal{R}\ T$, $\ldots$, $L'_n[S']\,\mathbf{,}S''\ \mathcal{R}\ T$ with $\bigcup_{j\in\{1,\ldots,n\}} \widehat{L'_j} = \widehat{L}$. This, by item 4.($b$), reduces to demonstrate that, for every $j \in \{1,\ldots,n\}$, $T \downarrow L[T']\,\mathbf{,}T''$ and $\widehat{L'_j} \subseteq \widehat{L}$ and $S'\ \mathcal{R}\ T'$ and $S''\ \mathcal{R}\ T''$. It follows that $T \downarrow L_i[T'_i]\,\mathbf{,}T''_i$ and $\widehat{L} \subseteq \bigcup_{i\in\{1,\ldots,n\}} \widehat{L_i}$ and $S'\mathcal{D}T'_i$ and $S'\mathcal{D}T''_i$ for some i. That is item 4 of $\mathcal{D}$. $\square$

A consequence of Lemma 2.6 is that the algorithm $S \preceq_{\mathtt{A}} T \Rightarrow \mathtt{A}'$, can be simplified into $S \preceq^{\mathtt{d}}_{\mathtt{A}} T \Rightarrow \mathtt{A}'$ where rules (SPLIT) and (LSEQ) are replaced by the rule:

(LAB)

$$\frac{(T \downarrow L_i[T_i]\,\mathbf{,}T'_i)^{i\in\{1,\ldots,n\}} \quad \widehat{L} \subseteq \bigcup_{i\in 1,\ldots,n} \widehat{L_i} \quad (\widehat{L} \cap \widehat{L_i} \neq \emptyset)^{i\in\{1,\ldots,n\}} \qquad S \preceq^{\mathtt{d}}_{\mathtt{A}_{i-1}} T_i \Rightarrow \mathtt{A}'_i \quad S' \preceq^{\mathtt{d}}_{\mathtt{A}'_i} T'_i \Rightarrow \mathtt{A}_i}{L[S]\,\mathbf{,}S' \preceq^{\mathtt{d}}_{\mathtt{A}_0} T \Rightarrow \mathtt{A}_n}$$

This diminishes significantly the computational complexity of the algorithm which can be proved to be quadratic in the size of the schemas S and T.

2.7.1 The code and its computational complexity

The pseudo-code of the algorithm for $S \preceq^{\mathtt{d}}_{\mathtt{A}} T \Rightarrow \mathtt{A}'$ is shown in Table 2.3. It is a boolean function using two associative tables `At`, and `Af`. `At`, similarly to `A`, stores schemas whose subschema relation is either verified or is being verified. `Af` stores schemas whose subschema relation have been already verified to be false. This improves the efficiency by

preventing that the same subschema relation is verified twice. Finally, an auxiliary function `Handles` returning the set of handles of a given schema $handles(S)$ is used. We do not detail its implementation because it is immediate by Definition 2.1. However we notice that the complexity of a naive implementation is $O(\|R\|)$.

The algorithm is initially invoked with every entry of `At` and `Af` set to `false`, – it computes $S \preceq^{\mathtt{d}}_{\emptyset} T$. It primarily verifies either if the subschema relation has been already computed (checking `At`[S][T] and `Af`[S][T]) or if the schema S does not retain any handle (`Handles`$(S) = \emptyset$); in these cases returns immediately. Then, if `Handles`$(T = \emptyset$, since `Handles`$(S) \neq \emptyset$, it answers `false`. Otherwise `At`[S][T] is set to `true`, meaning that the pair (S,T) is being verified, and begins the syntax-directed case analysis of the schemas as explained in Table 2.2. If the verification fails `At`[S][T] is replaced by `false` and `Af`[S][T] is set to `false`. The following Proposition shows that the algorithm is polynomial in the size of the schemas.

Proposition 2.6 `SubSchema` *terminates in polynomial time.*

Proof: Let $\mathfrak{t}(S)$ be the set of subterms of a schema S and let $\mid \cdot \mid$ be the cardinality function. The dimensions of the arrays `At` and `Af` is $(2\times \mid \mathfrak{t}(S) \cup \mathfrak{t}(T) \mid)^2$. The reason is due to the following facts:

1. the contravariance of $\langle\cdot\rangle^{\circ}$ may reduce $S <: T$ to $T' <: S'$ where $T' \in \mathfrak{t}(T)$ and

2. for any subterm in $\mathfrak{t}(S) \cup \mathfrak{t}(T)$ we may need a new term – a union schema $Q = T_1 + \cdots + T_n$ – where every T_i is a subterm of either S or T.

Let $\lceil$`true`$\rceil = 1$ and $\lceil$`false`$\rceil = 0$. Let $\mathtt{At}_i$ and $\mathtt{Af}_i$ denote the arrays `At` and `Af` when one of them has been modified exactly i times. The following invariants are preserved at the end of every line of `SubSchema`:

1. for every S,T: `(At[`S`][`T`] == false) or (Af[`S`][`T`] == false)`, that is `true` is never stored both in `Af`[S][T] and in `At`[S][T];

2. for every S,T: if ($\mathtt{Af}_i$[S][T] `== true`) then ($\mathtt{Af}_{i+1}$[S][T] `== true`), that is `true` is never deleted from `Af`;

3. $\sum_{S,T} \lceil \mathtt{At}_i[S][T] \rceil + \lceil \mathtt{Af}_i[S][T] \rceil \leq \sum_{S,T} \lceil \mathtt{At}_{i+1}[S][T] \rceil + \lceil \mathtt{Af}_{i+1}[S][T] \rceil$ (i.e. the total number of `true`s either grows or remains the same)

4. if $\sum_{S,T} \lceil \texttt{At}_i\texttt{[}S\texttt{][}T\texttt{]} \rceil + \lceil \texttt{Af}_i\texttt{[}S\texttt{][}T\texttt{]} \rceil = \sum_{S,T} \lceil \texttt{At}_{i+1}\texttt{[}S\texttt{] [}T\texttt{]} \rceil + \lceil \texttt{Af}_{i+1}\texttt{[}S\texttt{] [}T\texttt{]} \rceil$ then $\sum_{S,T} \lceil \texttt{Af}_i\texttt{[}S\texttt{] [}T\texttt{]} \rceil < \lceil \texttt{Af}_{i+1}\texttt{[}S\texttt{][}T\texttt{]} \rceil$ (i.e. when the total number of `trues` remains the same then the `trues` in `Af` strictly increase).

We observe that, in the worst case, the algorithm terminates when $\sum_{S,T} \lceil \texttt{At[}S\texttt{] [}T\texttt{]} \rceil + \lceil \texttt{Af[}S\texttt{][}T\texttt{]} \rceil$ is equal to $(2\times \mid \mathfrak{t}(S) \cup \mathfrak{t}(T) \mid)^2$. Invariants 3 and 4 guarantee terminations (the number of `trues` either grows or remains the same for at most $(2\times \mid \mathfrak{t}(S) \cup \mathfrak{t}(T) \mid)^2$ times before terminating). Invariants 1 and 2 state that `true` is never set in the same entry twice and it is never assigned to the same entry of the two arrays. Therefore, there may be at most $(2\times \mid \mathfrak{t}(S) \cup \mathfrak{t}(T) \mid)^2$ stores of `true` into `At` and each `true` may be "moved" at most once into `Af`. The cost of this movement is linear in the complexity of computing the set of handles of a schema. Since such a computation is in $O(\mid \texttt{subt}(S) \mid)$ (Proposition **??**), the total cost of `SubSchema` is $O(\mid \mathfrak{t}(S) \cup \mathfrak{t}(T) \mid^3)$. □

2.8 Conclusion

The schema language and its subschema relation, have been implemented in PiDuce – a concurrent language with native XML data types and pattern matching. In particular, the design of the PiDuce datatype and pattern languages, as well as most of the algorithms regarding these features, have been strongly influenced by the XDuce and CDuce prototypes – two functional languages with native XML datatypes. Similarly to XDuce and CDuce, the PiDuce compiler performs a semantic analysis guaranteeing that invalid documents can never be produced. For instance the following PiDuce program

```
import Google {
    doSpellingSuggestion: ⟨key[string],phrase[string],⟨return[string]⟩^O⟩^O
}  location="http://api.google.com/GoogleSearch.wsdl"
in new stdout : ⟨return[string]⟩^O
in Google.doSpellingSuggestion!( key["xxxxxx"],phrase["heello"],stdout )
```

imports the operation `doSpellingSuggestion` of the `GoogleWebService` and prints the result of the invocation – that is < `return` > `hello` < `/return` > – to the standard output. At compile time – the typechecker – downloads the WSDL interfaces located at the specified URI and verifies that the service provides the requested operation with a compatible

Table 2.3: The Subschema Algorithm for determined schemas

```
1  bool SubSchema(S, T, At, Af)          //Assume ℰ globally defined
2   bool res = false;
3   if At[S][T] res:= true;
4   else if Af[S][T] res:= false;
5   else if Handles(S) = ∅ res:= true;
6   else if Handles(T) = ∅ res:= false;
7   else
8    At[S][T]:= true;
9    match S with
10     () → res:= () ∈Handles(T);
11     R + R' → res:= SubSchema(R, T, At, Af)∧SubSchema(R', T, At, Af)
12     (R + R'),R'' → res:= SubSchema(R,R'', T, At, Af)
13                          ∧SubSchema(R',R'', T, At, Af)
14     U → res:=SubSchema(ℰ(U), T, At, Af)
15     U,S → res:=SubSchema(ℰ(U),S, T, At, Af)
16     ⟨R⟩^O,R' → let {T_1,...,T_n}:= {T' | ⟨Q⟩^κ, T' ∈Handles(T)∧ κ ≤ O
17                                        ∧ SubSchema(Q, R, At, Af)}
18                in let Q:= T_1 + ··· + T_n
19                in res:= SubSchema(R', Q, At, Af);
20     ⟨R⟩^I,R' → let {T_1,...,T_n}:= {T' | ⟨Q⟩^κ, T' ∈Handles(T) ∧ κ ≤ I
21                                        ∧ SubSchema(R, Q, At, Af)}
22                in let Q:= T_1 + ··· + T_n
23                in res:= SubSchema(R', Q, At, Af);
24     ⟨R⟩^IO,R' → let {T_1,...,T_n}:= {T' | ⟨Q⟩^κ, T' ∈Handles(T) ∧ κ ≤ IO
25                                        ∧ SubSchema(Q, R, At, Af)
26                                        ∧ SubSchema(R, Q, At, Af)}
27                 in let Q:= T_1 + ··· + T_n
28                 in res:= SubSchema(R', Q, At, Af);
29     {(m_h:S_h)^{h∈H}}, R' → let {T_1,...,T_n}:=
30                     {T' | {(n_j:T_j)^{j∈J}}, T' ∈Handles(T) ∧
31                            ∀n_j∃m_h : n_j = m_h ∧ SubSchema(S_h, T_j, At, Af)}
32                 in let Q:= T_1 + ··· + T_n
33                 in res:= SubSchema(R', Q, At, Af);
34     L[R],R' → let {L_1[Q_1],Q'_1,...,L_n[Q_n],Q'_n}:=
35                   {L'[Q],Q' | L'[Q],Q' ∈Handles(T) and L ∩ L_i ≠ ∅}
36                in res:= L ⊆ ⋃_{i∈{1,...,n}} L_i ∧ ⋀_{i=1..n} (SubSchema(R,R_i, At, Af)
37                                      ∧SubSchema(R',Q_i, At, Af));
38     //default is for ⟨S⟩^κ, {(m_h:S_h)^{h∈H}}, L[S]
39     default → res:= SubSchema(S,(), T, At, Af)
40  if (¬res) At[S][T]:= false; Af[S][T]:= true;
41  return res;
```

schema. Thus the typechecker verifies the schema of the XML value

$$\texttt{key}[\text{"xxxxxx"}]\texttt{,phrase}[\text{"heello"}]\texttt{,stdout.print}$$

to be a subschema of the schema that can be accepted by the selected operation.

With respect to XDuce and CDuce, a major technical difficulty in the PiDuce datatype language is that values, as in XML, may contain references that are URIs where values can be sent. A reference is represented by the WSDL interface describing the schema of the values it accepts. The semantics rules of PiDuce expose an environment that is partially supplied by local service declarations and partially by the global environment. The maintenance of this environment means that communications also gather information about the schemas of the references contained in the message. A related problem is found in the algorithm matching a document against a pattern (pattern matching). The algorithm checks if the document conforms with the schema specified in the pattern and returns a set of variable bindings. As in XDuce, pattern matching in PiDuce is implemented using top-down tree automata, but the presence of references inside values increases the complexity of the algorithm. In particular, verifying that a reference matches a pattern reduces to checking whether the schema of the reference is a subschema of the one specified in the pattern or not. This, in general, requires exponential time in the size of the tree automata of the pattern and may significantly degrade the run-time efficiency of possible implementations. To alleviate this problem we have shown a reasonable restriction bearing a polynomial algorithm. As an example, consider the PiDuce fragment:

```
new PrinterWebService { printJPeg : ⟨((⟨Pdf⟩⁰ + ⟨JPeg⟩⁰), JPeg⟩⁰ }
in PrinterService.printJPeg?*( printer:(⟨Pdf⟩⁰ + ⟨JPeg⟩⁰), fileJPeg:JPeg)
in match printer with
 ⟨JPeg⟩⁰ → printer!( fileJPeg )
 | ⟨Pdf⟩⁰ → printer!( convertToPdf(fileJPeg) )
```

It defines the PiDuce printer Web service that provides one operation `printJPeg` that is waiting for pairs containing the printer service and the image to be printed (the `*` says that the input on the reference `printJPeg` is replicated). Depending on the schema of the printing service – bound to the variable `printer` – one of the two branches is selected converting the image to a `Pdf` if needed. It stands that, in order to select the proper branch, the WSDL of the received reference is downloaded and the subschema relation is invoked.

We also note that PiDuce is only an example of how to use our schema language and our subschema relation. Other programming languages, such as BPEL and WSCDL

implementations, may be equipped with a similar type system in order to obtain statically typed Web services and statically typed programs interacting with Web services.

Chapter 3

Encoding of WSDL interfaces

In this chapter we compare the expressivity our schema language with state of art technologies for describing Web services interfaces: WSDL and XML-SCHEMA. In particular we provide an encoding of WSDL and XML-SCHEMA in our schema language. It stands that the subschema relation is suitable to be used in querying services repositories: a query asking for a service accepting a certain schema may safely returns any service with a "greater" schema.

Structure of the chapter Section 3.1 introduces services interfaces and services repositories. Section 3.2 introduces WSDL defining its syntax and presenting the encoding in our schema language. Similarly, Section 3.3 introduces XML-SCHEMA, its syntax and the encoding in our schema language. Some final remarks are given in Section 3.4.

3.1 Introduction

Web services are described by public interfaces written in the Web Services Description Language (WSDL). Through WSDL, a designer describes the programming interface of a Web service. This interface is specified in terms of operations supported by the Web service, where each operation could take one message as input and eventually return another message as output. A WSDL file describing an interface can be compiled into appropriate programming language to generate intermediate layers, called stubs, that make calls to the Web services transparent. Stubs establish the connection to the Web

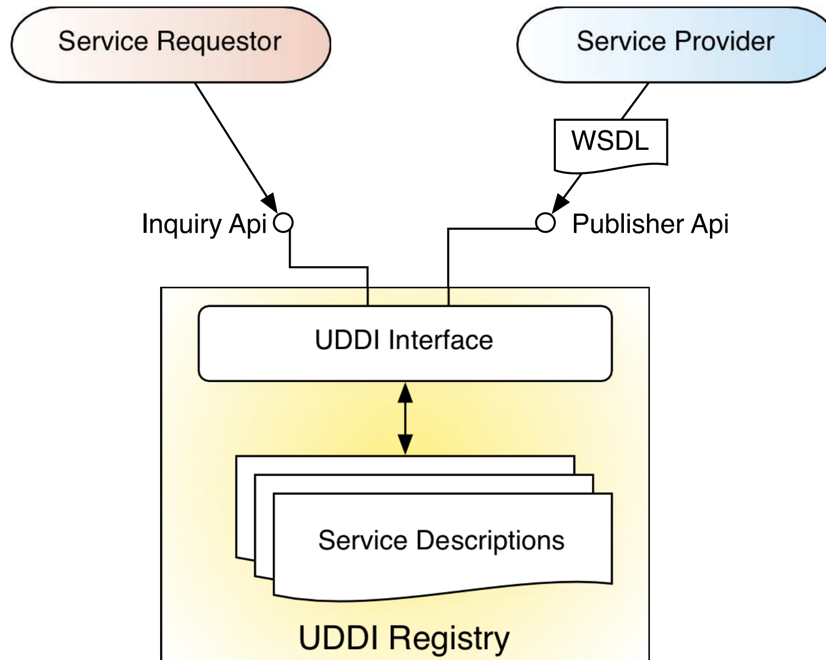


Figure 3.1: Interaction with a UDDI registry.

service selecting the appropriate communication protocol (typically HTTP) and selecting the appropriate message protocol (typically SOAP).

Once services have been described and can be invoked, services may be published in a repository. In this way requesters can look for services of interest and understand their properties i.e. their WSDL interface and the URI at which they are made available (Figure 3.1). Hence, a basic step toward having a functional Web services architecture is to standardize the Web service registry. Such a standardization is taking place as part of the *Universal Description, Discovery, and Integration* (UDDI) [Spe02] project. The registry is the place where service descriptions are published in catalogs that can be searched by users. The standardized UDDI APIs determine how to publish a service, and how to query the registry. UDDI registries maintain different access points URIs and different APIs for requesters, publishers, and other registries (UDDI registry may interact each other for replication and custody purposes).

Another attempt to standardize service repositories is WSIL [BBM+01, Cov02]. Both WSIL and UDDI assist in the publishing and discovery of services, but their models are distinctly different. UDDI implements service discovery using a centralized model of one

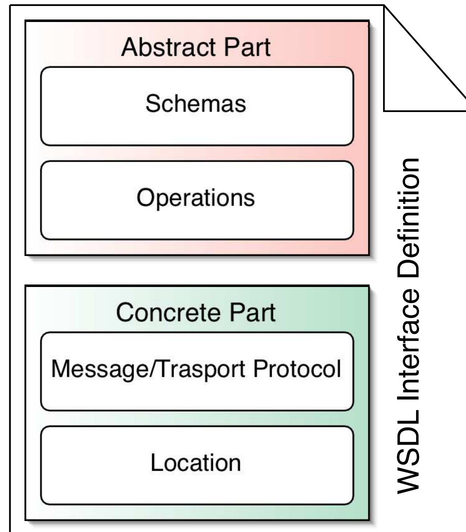


Figure 3.2: The structure of a WSDL.

or more repositories containing information on multiple business entities and the services they provide (UDDI is a sort of Yellow Pages service where multiple businesses are grouped and listed with a description of the goods or services they offer and how to contact them). WSIL approaches service discovery in a decentralized fashion, where service description information can be distributed to any location using a simple extensible XML document format. Unlike UDDI, WSIL does not concern itself with business entity information, nor does it specify a particular service description format. WSIL works under the assumption that you are already familiar with the service provider, and relies on other service description mechanisms such as WSDL.

Either UDDI and WSIL repositories may be queried by clients looking for services with specific functionalities [CJ04]. The current search mechanism can focus on several criteria. One these criteria is the service schema but a schema name matching is applied. With this respect our subschema relation is anatural way for adding more flexibility to this search mechanism.

3.2 WSDL interfaces

The Web Services Description Language (WSDL) is an XML format for describing Web services. Version 1.1 is currently widely used even if it has not been endorsed by the W3C. Version 2.0, for which several drafts have been released, is expected to become a W3C Recommendation and now it is in the Proposed Recommendation state. WSDL represents a contract between the service requester and the service provider, in much the same way that a JAVA interface represents a contract between client code and the actual JAVA object. The crucial difference is that WSDL is platform and language independent. A client can locate a Web service and invoke any of its publicly available functions. With WSDL-aware tools, it is possible to automate this process, enabling applications to easily integrate new services with little or no manual code. WSDL therefore represents a cornerstone of the web service architecture, because it provides a common language for describing services and a platform for automatically integrating those services. It 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.

As shown by Figure 3.2, every WSDL interface consists of two parts. An *abstract part* that defines the set of operations supported by the service; a *concrete part*: which binds every operation to a concrete network protocol and to a concrete location (URI). The abstract part is made of *schemas* and *interfaces* (ports in WSDL1.1). Schemas are defined using XML-SCHEMA; interfaces are logical groupings of operations that are defined by a name, an interaction pattern, and the schema of messages for invoking the operations and receiving back results and faults. For instance a service with a single input-only operation (we are omitting some details of the WSDL document) may be described by the WSDL in Table 3.1. Indeed it defines the service specifying both abstract and concrete information. In this case the service is located at `http://example.com/ser- viceInOnly` and that it provides an operation called `opInOnly`. Such an operation can be invoked with documents of schema `InvokeScm` and it implements the `In-Only` *interaction pattern*. This means that `opInOnly` only accepts messages without sending back any result.

Table 3.1: A simplified WSDL interface.

```
< description >
  < type >
    < element name = "InvokeScm" type = "string"/ >
  < /type >
  < interface name = "serviceInOnly" >
    < operation name = "opInOnly" pattern = "In-Only" >
      < input Label = "In" element = "InvokeScm"/ >
    < /operation >
  < /interface >
  < service name = "serviceInOnly" >
    < interfaces name = "opInOnly" protocol = "http"/ >
    < endpoint location = "http : //example.com/serviceInOnly"/ >
  < /service >
< /description >
```

3.2.1 Syntax of WSDL interface definitions

We now formally define the syntax of WSDL and we show WSDL interfaces can be encoded in our schema language. The syntax assumes a infinite set of interface names ranged over by I, I_1, ..., an infinite set of operation names ranged over by op, op_1, ... For simplicity sake, we also assume a schema environment $\mathcal{E}$. Such an environment is defined in the `type` part of the WSDL document by means of a XML-SCHEMA declaration. In Section 3.3 we show how to encode XML-SCHEMA in our schema language thus obtaining the proper $\mathcal{E}$. A WSDL service is defined by the grammar in Table 3.2.

There are two syntactic categories. $Interfaces$ have a name, specified by the attribute `name`, and contain a list of supported operations. *Operations* may use one of the eight predefined interaction patterns [CHL+06]. Four interaction patterns are used for describing interaction started by clients and four are used for describing interactions started by servers. Whilst the utility of first kind of patterns is obvious we remark that also the second kind of interactions patterns are quite useful in practice. Indeed they are often used by Web-services, for describing how they will interact with clients especially in asynchronous communication. For instance, a server, may accept references where it is possible to notify messages of a given schema. Such a notification is then specified by an interaction pattern describing the communication from the perspective of the server.

Table 3.2: Grammar of WSDL interfaces.

Interface ::=	< interface name = "*I*" >	**service interface**
	Operation ... *Operation*	
	< /interface >	
Operation ::=		**service operation**
	< operation name = "*op*" pattern = "In-Only" >	
	< input Label = "In" element = "U"/ >	
	< /operation >	
|	< operation name = "*op*" pattern = "Robust-In-Only" >	
	< input Label = "In" element = "U"/ >	
	< outfault Label = "OutFault" element = "U"/ >	
	< /operation >	
|	< operation name = "*op*" pattern = "In-Out" >	
	< input Label = "In" element = "U"/ >	
	< output Label = "Out" element = "U"/ >	
	< outfault Label = "OutFault" element = "U"/ >	
	< /operation >	
|	< operation name = "*op*" pattern = "Out-Opt-In" >	
	< input Label = "In" element = "U" >	
	< output Label = "Out" element = "U" >	
	< /operation >	
|	< operation name = "*op*" pattern = "Out-Only" >	
	< output Label = "Out" element = "U"/ >	
	< /operation >	
|	< operation name = "*op*" pattern = "Robust-Out-Only" >	
	< output Label = "Out" element = "U"/ >	
	< outfault Label = "OutFault" element = "U"/ >	
	< /operation >	
|	< operation name = "*op*" pattern = "Out-In" >	
	< output Label = "Out" element = "U"/ >	
	< input Label = "In" element = "U"/ >	
	< infault Label = "InFault" element = "U"/ >	
	< /operation >	
|	< operation name = "*op*" pattern = "Out-Opt-In" >	
	< output Label = "Out" element = "U" >	
	< input Label = "In" element = "U" >	
	< infault Label = "InFault" element = "U"/ >	
	< /operation >	

We now describe interaction patterns one-by-one.

1. The `In-Only` pattern consists of exactly one asynchronous message sent by the client to the server for invoking the operation (it corresponds to the *one-way* pattern of WSDL1.1).

2. The `Robust-In-Only` pattern consists of one asynchronous message for invoking the operation. Differently from the `In-Only` pattern that cannot raise any fault, the invocation message may *trigger* a fault. Then a further schema for describing the fault message is used.

3. The `In-Out` pattern consists of two messages: one for invoking the operation and one for receiving back the message returned by the invocation. In case of faults the returned message may be *replaced* by the a fault message (it corresponds to the *request-response* pattern of WSDL1.1).

4. The `In-Opt-Out` pattern consists of two messages: one for invoking the operation and one for receiving back the result of the invocation. This pattern differs from the previous one because the returning message may be either missing or a fault triggered by the client invocation.

5. The `Out-Only` pattern consists of exactly one asynchronous message sent from the server to the client (it corresponds to the *notification* pattern of WSDL1.1).

6. The `Robust-Out-Only` pattern consists of one asynchronous message sent from the server to the client. Since such an invocation message may trigger a fault, a further schema for describing the fault message sent back to the server is used.

7. The `Out-In` pattern consists of two messages: one from the server to the client and another one from the client to the server. In case of faults the message returned by the client may be replaced by the a fault message (it corresponds to the *solicit-response* pattern of WSDL1.1).

8. The `Out-Opt-In` pattern consists of two messages: one from the server to the client and another one from the client to the server. As in the `In-Opt-Out`, the last message may be either missing or replaced by a fault message triggered by the server invocation.

It follows that there are two kind of faults. Those triggered by a message – that are used by the pattern `Robust-In-Only`, `Robust-Out-Only`, `In-Opt-Out` and `Out-Opt-In` – and those replacing the return message – that are used by the pattern `In-Out` and `Out-In` (`Out-Only` and `In-Only` cannot raise any fault). In case of triggered faults, a different communication channel may be used for sending the fault message. In case of fault replacing messages, the communication channel where the fault is send is the same channel used for receiving other messages. For instance, if an error occurs in an operation implementing the interaction pattern `In-Opt-Out`, a fault is sent to the client. However, the fault is not necessarily sent on the same communication channel of the output message and two different channels may be used. This differs from the pattern `In-Out` where, in case of faults, the fault message replaces the output message thus requiring the same channel to be used for results and faults. In the following section we will see how this aspect influences the encoding.

3.2.2 Encoding of WSDL interfaces

We now detail the encoding of WSDL interface definitions with our schema language. The encoding has been also implemented – for WSDL1.1 – in the current prototype of PiDuce. At this moment PiDuce encodes WSDL1.1 interfaces because it is the version of WSDL currently used by Web-services. This allows real experimentations of the PiDuce language and architecture. However, in this section, we discuss the encoding of the new WSDL specification for two reasons: WSDL2.0 will replace in the near future WSDL1.1; and WSDL1.1 can be considered as a subset of WSDL1.1 (even if the syntax is quite different).

We assume $\mathcal{E}$ such that $\mathtt{U} \in \mathtt{dom}(\mathcal{E})$ for any schema `U` appearing in the WSDL interface and `In`, `Out`, `InFault`, `OutFault` to be valid labels.

Encoding the `In-Only` interaction pattern An `In-Only` operation is encoded as:

$$\left[\!\!\left[\begin{array}{l} <\mathtt{operation\ name} = "op" \ \mathtt{pattern} = "\mathtt{In\text{-}Only}" > \\ \quad <\mathtt{input\ Label} = "\mathtt{In}" \ \mathtt{element} = "\mathtt{U}" / > \\ < /\mathtt{operation} > \end{array}\right]\!\!\right] = \langle \mathtt{In}[\mathtt{U}] \rangle^{\mathtt{O}}$$

Technically, the tag $<\mathtt{input\ message} = "\mathtt{S}" / >$ in the WSDL must be interpreted as a schema constructor collecting references that may be invoked with values of schema S, or with subsets of such values. Said otherwise, the constructor $<\mathtt{inputmessage} = "\ldots" / >$

behaves contravariantly with respect to the argument schema. In our notation the operation *op* has schema $\langle \mathtt{U} \rangle^{\mathtt{O}}$. The capability "O" constrains the client to use the reference for outputs only.

Encoding the Robust-In-Only interaction pattern A `Robust-In-Only` operation is encoded as:

$$\left[\!\left[\begin{array}{l} <\mathtt{operation\ name} = \text{"}op\text{"}\ \mathtt{pattern} = \text{"}\mathtt{Robust\text{-}In\text{-}Only}\text{"} > \\ \quad <\mathtt{input\ Label} = \text{"}\mathtt{In}\text{"}\ \mathtt{element} = \text{"}\mathtt{U}_1\text{"}/> \\ \quad <\mathtt{infault\ Label} = \text{"}\mathtt{InFault}\text{"}\ \mathtt{element} = \text{"}\mathtt{U}_2\text{"}/> \\ </\mathtt{operation}> \end{array}\right]\!\right]$$
$$= \langle \mathtt{In}[\mathtt{U}_1], \langle \mathtt{InFault}[\mathtt{U}_2] \rangle^{\mathtt{I}} \rangle^{\mathtt{O}}$$

In this case, if an error occurs, a fault message is sent back to the client. Operationally, this is equivalent to delivering, together with the message of schema $\mathtt{In}[\mathtt{U}_1]$ a fresh reference of schema $\langle \mathtt{InFault}[\mathtt{U}_2] \rangle^{\mathtt{I}}$ where the client reads for faults. Thus we obtain the schema $\langle \mathtt{In}[\mathtt{U}_1], \langle \mathtt{InFault}[\mathtt{U}_2] \rangle^{\mathtt{I}} \rangle^{\mathtt{O}}$ where the capability "O" constraints the client to use the reference for sending data and the capability "I" constrains the client to use the fresh reference for only for receiving fault messages.

Encoding the Out-Only interaction pattern An `Out-Only` operation is encoded as:

$$\left[\!\left[\begin{array}{l} <\mathtt{operation\ name} = \text{"}op\text{"}\ \mathtt{pattern} = \text{"}\mathtt{Out\text{-}Only}\text{"} > \\ \quad <\mathtt{output\ Label} = \text{"}\mathtt{Out}\text{"}\ \mathtt{element} = \text{"}\mathtt{U}\text{"}/> \\ </\mathtt{operation}> \end{array}\right]\!\right] = \langle \mathtt{Out}[\mathtt{U}] \rangle^{\mathtt{I}}$$

The intended meaning of this pattern is that the remote service is communicating the schema of the messages it will send back. To receive this message, the client service has to create a reference whose schema is (greater than) $\langle \mathtt{Out}[\mathtt{U}] \rangle^{\mathtt{I}}$.

Encoding the Robust-Out-Only interaction pattern A `Robust-Out-Only` operation is encoded as follows:

$$\left[\!\left[\begin{array}{l} <\mathtt{operation\ name} = \text{"}op\text{"}\ \mathtt{pattern} = \text{"}\mathtt{Robust\text{-}In\text{-}Only}\text{"} > \\ \quad <\mathtt{output\ Label} = \text{"}\mathtt{Out}\text{"}\ \mathtt{element} = \text{"}\mathtt{U}_1\text{"}/> \\ \quad <\mathtt{outfault\ Label} = \text{"}\mathtt{OutFault}\text{"}\ \mathtt{element} = \text{"}\mathtt{U}_2\text{"}/> \\ </\mathtt{operation}> \end{array}\right]\!\right]$$
$$= \langle \mathtt{Out}[\mathtt{U}_1], \langle \mathtt{OutFault}[\mathtt{U}_2] \rangle^{\mathtt{O}} \rangle^{\mathtt{I}}$$

In this case, if an error occurs sending the message, a fault is sent back to the server. This is equivalent to delivering, together with the message of schema $\texttt{Out}[\texttt{U}_1]$, a fresh reference of schema $\langle\texttt{OutFault}[\texttt{U}_2]\rangle^{\texttt{I}}$ where the server reads for faults.

Encoding the In-Out interaction pattern An In-Out operation is encoded as follows:

$$\left[\!\!\left[\begin{array}{l} <\texttt{operation name} = \text{"}op\text{"}\ \texttt{pattern} = \text{"}\texttt{In-Out}\text{"} > \\ \quad <\texttt{input Label} = \text{"}\texttt{In}\text{"}\ \texttt{element} = \text{"}\texttt{U}_1\text{"}/> \\ \quad <\texttt{output Label} = \text{"}\texttt{Out}\text{"}\ \texttt{element} = \text{"}\texttt{U}_2\text{"}/> \\ \quad <\texttt{outfault Label} = \text{"}\texttt{OutFault}\text{"}\ \texttt{element} = \text{"}\texttt{U}_3\text{"}/> \\ </\texttt{operation}> \end{array}\right]\!\!\right]$$
$$= \langle\texttt{In}[\texttt{U}_1], \langle\texttt{Out}[\texttt{U}_2] + \texttt{OutFault}[\texttt{U}_3]\rangle^{\texttt{I}}\rangle^{\texttt{O}}$$

In this case the service may be interpreted as a function. Then, together with the message of schema $\texttt{In}[\texttt{U}]$ is also delivered a channel with schema $\langle\texttt{Out}[\texttt{U}_2] + \texttt{OutFault}[\texttt{U}_3]\rangle^{\texttt{I}}$ where both, output messages and fault messages can be sent. The server triggers the client sending the proper data over such a reference while the client reads either return messages and faults.

Encoding the In-Opt-Out interaction pattern An In-Opt-Out operation is encoded as follows:

$$\left[\!\!\left[\begin{array}{l} <\texttt{operation name} = \text{"}op\text{"}\ \texttt{pattern} = \text{"}\texttt{In-Opt-Out}\text{"} > \\ \quad <\texttt{input Label} = \text{"}\texttt{In}\text{"}\ \texttt{element} = \text{"}\texttt{U}_1\text{"}/> \\ \quad <\texttt{output Label} = \text{"}\texttt{Out}\text{"}\ \texttt{element} = \text{"}\texttt{U}_2\text{"}/> \\ \quad <\texttt{outfault Label} = \text{"}\texttt{OutFault}\text{"}\ \texttt{element} = \text{"}\texttt{U}_3\text{"}/> \\ </\texttt{operation}> \end{array}\right]\!\!\right]$$
$$= \langle\texttt{In}[\texttt{U}_1], \langle\texttt{Out}[\texttt{U}_2]\rangle^{\texttt{I}}, \langle\texttt{OutFault}[\texttt{U}_3]\rangle^{\texttt{I}}\rangle^{\texttt{O}}$$

This case is similar to the previous one but two different channels can be used for reading the output message and the fault message. It is worth to notice the, thanks to our subschema relation, clients may safely communicate the same reference of schema $\langle\texttt{Out}[\texttt{U}_2] + \texttt{OutFault}[\texttt{U}_3]\rangle^{\texttt{I}}$ for reading both outputs and faults. Indeed $\langle\texttt{Out}[\texttt{U}_2] + \texttt{OutFault}[\texttt{U}_3]\rangle^{\texttt{O}}$ is a subschema of both $\langle\texttt{Out}[\texttt{U}_2]\rangle^{\texttt{O}}$ and $\langle\texttt{OutFault}[\texttt{U}_3]\rangle^{\texttt{O}}$.

Encoding the `Out-In` interaction pattern An `Out-In` operation is encoded as follows:

$$\left[\!\!\left[\begin{array}{l} <\texttt{operation name} = "op"\ \texttt{pattern} = "\texttt{Out-In}" > \\ \quad <\texttt{output Label} = "\texttt{Out}"\ \texttt{element} = "\texttt{U}_1"/> \\ \quad <\texttt{input Label} = "\texttt{In}"\ \texttt{element} = "\texttt{U}_2"/> \\ \quad <\texttt{infault Label} = "\texttt{InFault}"\ \texttt{element} = "\texttt{U}_3"/> \\ </\texttt{operation}> \end{array}\right]\!\!\right] \\ = \langle \texttt{Out}[\texttt{U}_1], \langle \texttt{In}[\texttt{U}_2] + \texttt{InFault}[\texttt{U}_3]\rangle^{\texttt{O}}\rangle^{\texttt{I}}$$

Also in this case two references are created during the connection. The first reference is for receiving solicitations and the second is for responses. Therefore, together with the message of schema $\texttt{Out}[\texttt{U}]$, a fresh reference of schema $\langle \texttt{In}[\texttt{U}_2] + \texttt{InFault}[\texttt{U}_3]\rangle^{\texttt{O}}$ is delivered. The client will trigger the server sending the proper data over such a fresh reference.

Encoding the `Out-Opt-In` interaction pattern An `Out-Opt-In` operation is encoded as follows:

$$\left[\!\!\left[\begin{array}{l} <\texttt{operation name} = "op"\ \texttt{pattern} = "\texttt{Out-Opt-In}" > \\ \quad <\texttt{input Label} = "\texttt{Out}"\ \texttt{element} = "\texttt{U}_1"/> \\ \quad <\texttt{output Label} = "\texttt{In}"\ \texttt{element} = "\texttt{U}_2"/> \\ \quad <\texttt{infault Label} = "\texttt{InFault}"\ \texttt{element} = "\texttt{U}_3"/> \\ </\texttt{operation}> \end{array}\right]\!\!\right] \\ = \langle \texttt{Out}[\texttt{U}_1], \langle \texttt{In}[\texttt{U}_2]\rangle^{\texttt{O}}, \langle \texttt{InFault}[\texttt{U}_3]\rangle^{\texttt{O}}\rangle^{\texttt{I}}$$

This case is specular to the case `In-Opt-Out`. Therefore two different channels can be used for messages and faults. As in the pattern `In-Opt-Out`, the same reference – with schema $\langle \texttt{In}[\texttt{U}_2] + \texttt{InFault}[\texttt{U}_3]\rangle^{\texttt{O}}$ – may be used by the server for both return messages and faults. This is safe because $\langle \texttt{In}[\texttt{U}_2] + \texttt{InFault}[\texttt{U}_3]\rangle^{\texttt{O}} <: \langle \texttt{In}[\texttt{U}_2]\rangle^{\texttt{O}}$ and $\langle \texttt{In}[\texttt{U}_2] + \texttt{InFault}[\texttt{U}_3]\rangle^{\texttt{O}} <: \langle \texttt{InFault}[\texttt{U}_2]\rangle^{\texttt{O}}$.

Encoding the service Let $\texttt{opName}(I)$ be the function returning the value of the attribute `name` of the interface I. A service interface is encoded as:

$$\left[\!\!\left[\begin{array}{l} <\texttt{interface name} = "int" > \\ \quad I_0\ \ldots\ I_n \\ <\texttt{interface}/> \end{array}\right]\!\!\right] = \{\texttt{op}(I_0) : [\![I_0]\!]\ \ldots\ \texttt{op}(I_n) : [\![I_n]\!]\}$$

3.3 XML-SCHEMA

XML-SCHEMA is probably the most used schema language. It is a W3C recommendation and it is widely used for defining WSDL interfaces. In this section we introduce XML-SCHEMA giving some simple examples including only the features which are relevant in the Web service context. For this reason:

- XML attributes have been ignored because they would have entangled our schemas without giving any substantial contribution to their semantic relevance;
- features such as keys, references, and facets have been ignored because they are used mainly for validation rather than for typechecking.

In both cases such features are unused in the description of existing Web services (in particular in the XML-SCHEMAs of the exchanged messages), hence omitting their treatment does not impede actual experimentation of our schema language. For simplicity sake we also omit substitution groups, element references and restriction on base types. However it is quite easy to deal with these aspects. The current prototype of PiDuce provides for their implementation.

Like all XML schema languages, XML-SCHEMA expresses sets of rules to which a XML document must conform in order to be considered *valid*. In particular XML-SCHEMA uses a XML syntax to define type of XML documents in terms of constraints upon what elements may appear, their relationship to each other, what types of data may be in them. For instance

```
< element name = "a "type = "int"/ >
```

defines documents having a as root and containing integers. Elements may be combined in sequences and choices. The XML-SCHEMA

```
< sequence minOccurs = "1" maxOccurs = "unbounded" >
 < element name = "a "type = "int"/ >
< /sequence >
```

defines a non empty sequence of a-labelled documents containing integers and

```
< choice minOccurs = "0" maxOccurs = "unbounded" >
  < element name = "a "type = "int"/ >
  < element name = "b "type = "string"/ >
< /choice >
```

defines any sequence of either *a* or *b* labelled documents containing integers and strings respectively. Thus the document <*a*> `1` < /*a*><*b*> `two` < /*b*><*a*> `3` < /*a*> is valid for such a schema. Names can be assigned to definitions by using complex types. For instance

```
< complexType name = "U" >
  < element name = "a "type = "int"/ >
< /complexType >
```

assigns the name `U` to the element *a*.

XML-SCHEMA provides two derivation mechanisms – *extension* and *restriction* – that are used for deriving schemas from other schemas. The derivation by extension is used for deriving schemas from other schemas appending elements. As an example consider the XML-SCHEMA fragment

```
< complexType name = "U" >
 < element name = "a "type = "int"/ >
< /complexType >
```

```
< element name = "b" >
 < extension base = "U" >
  < element name = "c "type = "string"/ >
 < /extension >
< /element >
```

It says that an element *b* is valid only if it contains a document of schema `U` followed by an element *c* containing strings. Then, the content of the element *b* has not schema `U`. Indeed a value <*a*> `5` < /*a*><*c*> `hello` < /*c*> is not valid for `U` because of the element *c*. The derivation by restriction restricts the content of the base schemas. A schema derived by restriction is similar to its base schema, except that its declarations are more limited than the corresponding declarations in the base type. In other words, the values represented by the new schema are a subset of the values represented by the base schema (as is the case with restriction of simple types). Therefore an application prepared for the values of the base type would not be surprised by the values of the restricted type. For example the definition

```
< complexType name = "U" >
  < sequence minOccurs = "0" maxOccurs = "1" >
    < element name = "a" type = "int"/ >
  < /sequence >
< /complexType >
```

```
< element name = "b" >
  < restriction base = "U" >
    < sequence minOccurs = "1" maxOccurs = "1" >
      < element name = "a" type = "int"/ >
    < /sequence >
  < /restriction >
< /element >
```

says that an element b is valid only if it contains a document labeled with a and containing integers. The content of b restricts the schema U because, in U, the element a is required whilst in b it is optional. Differently from the derivation by extension, in derivation by restriction the content of b is valid for the schema U. Indeed the documents with a as root and containing integers are always valid for U. Hence, we have that in derivation by restriction we obtain a schema which is in a subschema relation with its base schema. This is not true for derivations by extension.

3.3.1 Syntax of XML-SCHEMA

We assume two disjoint countably infinite sets: the *tags*, ranged over by $a, b, \cdots \in \mathcal{L}$, and the XML-SCHEMA *names*, ranged over by $\mathtt{U}, \mathtt{V}, \cdots \in \mathcal{N}$. We also assume B as base types (Int, String, n, s). The syntax of XML-SCHEMA definitions are defined by the grammar in Table 3.3 A XML-SCHEMA definition Xsd is a, possibly empty, sequence of either: element definitions E or complex type definitions. Complex types definitions are defined by a unique name U and a content model X. For instance

```
< complexType name = "U" >
  < element name = "a" type = "int"/ >
< /complexType >
```

assigns the schema name U to its content model: the element a containing integers.

Elements E can have different contents: a simple content specified by a base type B; a complex content specified by the complex type U; an inner defined complex content X; a reference content specified by a Web-service interface definition. This last possibility was introduced in WSDL2.0 and allows to communicate typed references. However, the finer entity that can be communicated, is the service and not the single operation.

The content models X are sequences of contents, choices of contents, single elements, repetitions, extensions, and restrictions. The repetition allow us to define sequences spec-

Table 3.3: Grammar of XML-SCHEMA definitions.

Xsd ::=		**definition**
	| ϵ	(empty)
	| $E\ Xsd$	(element)
	| < `complexType` `name` = ”`U`” >X< /`complexType` > Xsd	(complex type)
E ::=		**element**
	| < `element` `name` = ”a ”`type` = ”`B`” >	(base)
	| < `element` `name` = ”a ”`type` = ”`U`”/ >	(complex ref)
	| < `element` `name` = ”a ” >X< /`element` >	(complex)
	| < `element` `name` = ”a ” >$Interface$< /`element` >	(interface ref.)
X ::=		**content**
	| ϵ	(empty)
	| E	(element)
	| < `sequence` >$X_1\ \ldots\ X_n$< /`sequence` >	(sequence)
	| < `choice` >$X_1\ \ldots\ X_n$< /`choice` >	(choice)
	| <*particle* `minOccurs` = ”n” `maxOccurs` = ”m” >	(repetition)
	| X< /*particle*> *particle* $\in$ {`sequence`, `choice`}	
	| < `extension` `base` = ”`U`” >X< /`extension` >	(extension)
	| < `restriction` `base` = ”`U`” >X< `restriction` >	(restriction)

ifying the occurrences of the content through the attributes `minOccurs` and `maxOccurs`. Of course the value of the attribute `minOccurs` must be less then the value of the attribute `maxOccurs` ($n \leq$ `unbounded` for every n).

We note that, in XML-SCHEMA, non-regular languages cannot be defined because constant schema names are always underneath an *element* constructor. However, with this syntactic constraint, horizontal recursion cannot be defined using schema names. For this reason XML-SCHEMA provide the “$*$” operator through repetition schemas with `maxOccurs` set to `unbounded`.

3.3.2 Encoding of XML-SCHEMA

The correspondence between our schema language and XML-SCHEMA is established by suitable encoding function. This function is important because, as we said, XML-SCHEMA is the W3C schema language that is used in WSDL interfaces. The encoding is defined in

Table 3.4: Decoding of XML-SCHEMA.

$$
\begin{array}{ll}
[\![\epsilon]\!]_{\mathcal{E}} & = \mathcal{E} \\
[\![<\texttt{complexType name} = \text{"}\texttt{U}\text{"} > X \quad </\texttt{complexType}> \; Xsd]\!]_{\mathcal{E}} & = [\![Xsd]\!]_{\mathcal{E}} + [\texttt{U} \mapsto [\![X]\!]\,] \\
[\![<\texttt{element name} = \text{"}a\text{"}\,\texttt{type} = \text{"}\texttt{B}\text{"}/>]\!] & = a[\texttt{B}] \\
[\![<\texttt{element name} = \text{"}a\text{"}\,\texttt{type} = \text{"}\texttt{U}\text{"}/>]\!] & = a[\texttt{U}] \\
[\![<\texttt{element name} = \text{"}a\text{"} > X </\texttt{element}>]\!] & = a[\,[\![X]\!]\,] \\
[\![<\texttt{element name} = \text{"}a\text{"} > Interface </\texttt{element}>]\!] & = a[\,[\![Interface]\!]\,] \\
[\![\epsilon]\!] & = \texttt{()} \\
[\![<\texttt{sequence}> X_1 \; \ldots \; X_n </\texttt{sequence}>]\!] & = [\![X_1]\!]\texttt{,} \; \ldots \; \texttt{,}[\![X_n]\!] \\
[\![<\texttt{choice}> X_1 \; \ldots \; X_n </\texttt{choice}>]\!] & = [\![X_1]\!] + \; \ldots \; + [\![X_n]\!] \\
[\![<particle\ \texttt{minOccurs} = \text{"}m\text{"}\ \texttt{maxOccurs} = \text{"}n\text{"} > X \quad </particle>]\!] & = [\![X]\!]^{m,n} \\
[\![<\texttt{extension base} = \text{"}\texttt{U}\text{"} > X </\texttt{extension}>]\!] & = \texttt{U,}[\![X]\!] \\
[\![<\texttt{restriction base} = \text{"}\texttt{U}\text{"} > X </\texttt{restriction}>]\!] & = [\![X]\!]
\end{array}
$$

where:

$$
\begin{array}{lcl}
S^{0,0} & = & \texttt{()} \\
S^{0,n+1} & = & \texttt{()} + S\texttt{,}S^{0,n} \\
S^{m+1,n+1} & = & S\texttt{,}S^{m,n} \\
S^{0,\text{unbounded}} & = & S^{*} \\
S^{m+1,\text{unbounded}} & = & S\texttt{,}S^{m,\text{unbounded}}
\end{array}
$$

Table 3.4 where

$$
(\mathcal{E} + [\texttt{U} \mapsto S])(\texttt{V}) \stackrel{\text{def}}{=} \begin{cases} \mathcal{E}(\texttt{V}) & \text{if } \texttt{V} \neq \texttt{U} \\ S & \text{otherwise} \end{cases}
$$

Thus, the encoding is a function $[\![\cdot]\!] : (Xsd, \mathcal{E}) \to \mathcal{E}$ taking a XML-SCHEMA definition and a schema environment as inputs and returning a new environment $\mathcal{E}$ containing the complex types defined in the Xsd. The encoding function is quite straightforward. XML-SCHEMA elements are encoded as elements of our schema language. Sequences and choices are encoded with the correspondent operators "," and "+". Repetitions are encoded using an auxiliary repetition operator over schema $S^{n,m}$. For instance $a[\;]^{0,0} = \texttt{()}$, $a[\;]^{1,\text{unbounded}} = a[\;]\texttt{,}a[\;]^{*}$, and $a[\;]^{1,2} = a[\;], (a[\;] + \texttt{()})$. It is worth to notice that expanding $a[\;]^{1,2}$ with $a[\;] + a[\;]\texttt{,}a[\;]$ is also reasonable but it leads to a non labeled-determined schema.

In case of extensions the encoding of the content is appended to the encoding of

the base schema. Since the complex type U is guarded the resulting schema is valid ($\texttt{ntail}(\texttt{U}) = \emptyset$). In case of restrictions the encoded schema is the encoding of the content. By the XML-SCHEMA specification [Gro04c] (paragraph 2.2.1.1) restrictions might include narrowed ranges or reduced alternatives. Whilst it is clear that *reduced alternatives* imply the schemas to be in the subschema relation ($S <: S + T$) it may be not clear that the same holds for *narrowed ranges*. This is what the next Lemma 3.1 states.

Lemma 3.1 *If* $n < n'$ *then* $S^{m,n} <: S^{m,n'}$.

Proof: We prove the Lemma by induction m.

– if ($m = 0$) we proceed by induction on n. If $n = 0$ we have to show that $S^{0,0} <: S^{0,n'}$ with $S^{0,0} = \texttt{()}$. We distinguish two cases. If $n' = \texttt{unbounded}$ then $S^* = \texttt{U}$ with $\mathcal{E}(\texttt{U}) = S\texttt{,}S^* + \texttt{()}$ and we conclude. If $n' \neq \texttt{unbounded}$ then $S^{0,n'} = \texttt{()} + S^{0,n'-1}$ and we conclude.

  If $n > 0$ we have to show that $S^{0,n} <: S^{0,n'}$ with $S^{0,n} = \texttt{()} + S\texttt{,}S^{0,n-1}$. We distinguish two cases. If $n' = \texttt{unbounded}$ then $S^* = \texttt{U}$ with $\mathcal{E}(\texttt{U}) = S\texttt{,}S^* + \texttt{()}$. We conclude by the inductive hypothesis $S^{0,n-1} <: S^*$. If $n' \neq \texttt{unbounded}$ then $S^{0,n} = \texttt{()} + S^{0,n-1}$ and $S^{0,n'-1} = \texttt{()} + S^{0,n'-1}$ we conclude by inductive hypothesis.

– if ($m > 0$), since $m < n$, we also have $n > 0$. We distinguish two cases. If $n' = \texttt{unbounded}$, we have to show $S^{m,n} <: S^{m,\texttt{unbounded}}$ with $S^{m,n} = S\texttt{,}S^{m-1,n-1}$. Since $S^* = \texttt{U}$ with $\mathcal{E}(\texttt{U}) = S\texttt{,}S^* + \texttt{()}$ we conclude by the inductive hypothesis $S^{m-1,n-1} <: S^*$. If $n' \neq \texttt{unbounded}$, we have to show that $S^{m,n} <: S^{m,n'}$ with $S^{m,n} = S\texttt{,}S^{m-1,n-1}$ and $S^{m,n'} = S\texttt{,}S^{m-1,n'-1}$. We conclude by the inductive hypothesis $S^{m-1,n-1} <: S^{m-1,n'-1}$. $\square$

Label-Determined schemas In XML-SCHEMA, when a particle contains identically named element declarations, the type definitions of those declarations must be the same.

For example the definition:

```
< element name = "a" >
  < choice >
    < sequence >
      < element name = "b "type = "U"/ >
      < element name = "c "type = "int"/ >
    < /sequence >
    < element name = "b "type = "V"/ >
  < /choice >
< /element >
```

is valid only if $\mathtt{U} = \mathtt{V}$. The encoding of these kind of declarations is the only way of producing non label-determined schemas. Indeed we obtain $a[b[\mathtt{U}]\mathtt{,}c[\mathtt{Int}] + b[\mathtt{V}]\mathtt{,()}]$ that it is not labelled-determined. However, since $\mathtt{U} = \mathtt{V}$, by Lemma 2.2 of Chapter 2 we can always rewrite such schemas into an equivalent label-determined schema that is $b[\mathtt{U}]\mathtt{,}(c[\mathtt{Int}] + \mathtt{()})$ in the example above.

3.4 Conclusions

The encoding of WSDL and XML-SCHEMA have been implemented in the current prototype of PiDuce and experimentation with many Web services such as Google, MSN, and Amazon have been successfully performed. Indeed, one of the main purpose of the PiDuce project is to design an distributed machine running applications that may be exported to the Web (Web services) and that may import and compose external Web services. Thus interoperability is one of the main effort in the implementation. PiDuce imports WSDL and the related XML-SCHEMA converting them into the PiDuce type system that is the schema language and the subschema relation of the Chapter 2. This operation is used at compile time and at runtime. At compile time, the WSDL is encoded for typechecking. This ensures that external Web services are invoked with the proper data thus avoiding runtime type errors. At runtime the encoding is used for validating XML documents coming from other parties and for pattern matching purposes. Indeed, every time that a XML document is received it is verified that it conforms with the schema of the reference and, in negative case, the document is discarded. This conformity check amounts to infer the schema of the document and to verify that it is a subschema of the schema of the reference. In case of document containing references, the schema of the document is determined downloading

the WSDL interface of the references therein. In order to keep low the cost of subschema relation invoked at runtime, the PiDuce compiler gives a warning whenever a non labeled-determined schema is used for references. If no warnings are given then the subschema relation will be invoked on label-determined schemas only, thus reducing to the algorithm in Table 2.3 of Chapter 2 and guaranteeing a polynomial complexity.

Part III

Behavioral contracts

Chapter 4

The Finite Contract Language

In this chapter we define a formal – finite – contract language along with subcontract and compliance relations. We then extrapolate contracts out of processes, that are recursion-free fragment of CCS. We finally demonstrate that a client completes its interactions with a service provided the corresponding contracts comply. Our contract language may be used as a foundation of Web services technologies, such as WSDL message exchanges patterns and WSCL.

Structure of the chapter. Section 4.1 is an introduction to the contract language. Section 4.2 referes to some related works. Section 4.3 formally defines the contract language along with *subcontract* and *compliance* relations. In Section 4.4 we relate the language with WSDL message exchange patterns that are used to to specify simple service protocols. Our notion of compliance between contracts is lifted to a notion of *compliance* between processes in Section 4.5 and the correspondence between the two relations is shown.

4.1 Introduction

Web services are loosely coupled software systems exchanging sequence of messages in a distributed and dynamic environment. In this context, it is fundamental for clients to be able to search – also at run-time – services with the required capabilities, namely the format of the exchanged messages, and the protocol – or *behavioral contract* – required

to interact successfully with the service. In turn, services are required to publish such capabilities in some known repository.

As we said in Chapter 3, the Web Service Description Language provides a standardized technology for describing the interface exposed by a service in terms of: physical location, the schema of the exchanged messages, the transport protocol to be used (i.e. HTTP, SMTP, or others), the message protocol (i.e. SOAP), and the interaction pattern. The subschema relation of Chapter 2 verifies the compatibility of data exchanged during communications. However the subschema relation does not take into account the conversation protocol implemented by the service. Indeed, WSDL interaction patterns are specifications of simple conversation protocols. A Web service operation specified by the `In-Out` interaction pattern, defines a conversation protocol where an input message on a certain name is performed and then either a fault message or the output message is sent back to the client. Thus, clients implementing the `Out-In` interaction pattern are compatible, in the sense that they complete their protocol. Similarly, clients implementing the `Out` interaction pattern are compatible because they also complete their protocol (even if the server does not). This kind of *protocol compatibility* is not considered by the subschema relation.

In this chapter we define a calculus for (behavioral) contracts along with a subcontract relation, and we formalize the relationship between contracts and processes (that is clients and services) exposing a given contract. Contracts are made of actions to be interpreted as either message types or communication ports. Actions may be combined by means of two choice operators: $+$ represents the *external choice*, meaning that the interacting part decides which one of alternative conversations to carry on; $\oplus$ represents the *internal choice*, meaning that the choice is not left to the interacting part. As a matter of facts, contracts are behavioral types of processes that do not manifest internal moves and the parallel structure. They are *acceptance trees* in Hennessy's terminology [Hen85, Hen88].

Then we devise a *subcontract relation* $\preceq$ such that a contract σ is a subcontract of σ' if σ manifests less interacting capabilities than σ'. The subcontract relation can then be used for querying (Web services) repositories. A query for a service with contract σ may safely return services with contract σ' such that $\sigma \preceq \sigma'$. It is possible that interaction with a service that exposes a contract that is bigger than the client requires may result into unused capabilities on the server side. We argue that this is safe, because we are interested in the client's ability to complete the interaction. Such client completion property inspires

a relationship between client contracts and service ones – the *contract completion* – that may be defined in terms of $\preceq$ and an appropriate complement operation over contracts.

To illustrate our contracts at work we consider a recursion-free fragment of the Calculus of Communicating Systems (CCS [Mil82]). We define a *compliance* relation between processes such that a process – the client – interacting with another – the service – is guaranteed to *complete*. For instance the clients $(a.b \mid \overline{a})\backslash a$ and $(a.b \mid a.c \mid \overline{a})\backslash a$ respectively comply with the services $\overline{b}$ and $\overline{b} \mid \overline{c}$; the two clients do not comply with $\overline{c}$. We then extrapolate a contract out of a process by means of a type system defined using the expansion theorem in [NH87]. For instance, we are able to deduce $a.b \mid \overline{a} \vdash (a.(b.\overline{a}+\overline{a}.b)+\overline{a}.(a.b+b.a)+b) \oplus b$. Finally we prove our main result: *if the contract of a client complies with the contract of a service, then the client complies with the service.*

4.2 Related work

The use of formal models to describe communication protocols is not new. For instance the SOAP Service Description Language (SSDL) [Sav05] is a XML language for describing Web Services contracts in terms of SOAP messages and protocols. The primary goal of a SSDL contract is to provide the mechanisms for service architects to describe the structure of the SOAP messages a Web Service supports. Once the messages of a Web Service have been described, the SSDL protocol framework – that is based on CSP [Hoa04] and on the π-calculus [Mil91] – can be used to combine the messages into protocols that expose the messaging behaviour of that Web service.

The *stuck free conformance* relation developed by Fournet et al. in [FHRR04, RR02] is inspired by the theory of refusal testing [Phi87] and is based on the notion of stuck-freedom system. Stuck-freedom formalizes the property that a communicating system cannot get stuck, either by ending in deadlock waiting for messages that are never sent, or by sending messages that are never received. Starting from the notion of stuck-freedom, the notion of stuck-free conformance is a refinement relation on CCS processes with the property that specifications can safely be substituted for implementations with respect to the stuck-freedom.

Session types developed by Honda et al. [THK94, HVK98] is an advanced type system where types – given to (session) channels – specify the sequence of message types sent

and received. Messages might be choices between a range of possibilities and such choices might be either internal or external, thus types have a branching structure. For instance $\&\{l_1 : \rho_1,\ \ldots, l_n : \rho_n\}$ waits with n options, and behaves as type ρ_i if the i-th action is selected (external choice). Similarly, $\oplus\{l_1 : \rho_1,\ \ldots, l_n : \rho_n\}$ represents the behavior which would select one of l_i and then behaves as ρ_i (internal choice). Gay and Hole in [GH99] add subtyping to session types and show the application of the resulting type system to client-server interactions. Carbone et al. in [CHYa, CHY$^+$b] present two paradigms of description of communication behavior – one focusing on global message flows and another on endpoint behaviors, as formal calculi based on session types.

The contract language we present is inspired by "CCS without τ" [NH87] and by Hennessy's model of acceptance trees [Hen85, Hen88]. To the best of our knowledge the subcontract relation $\preceq$ is original. It is incomparable with may testing preorder and it is less discriminating than the must testing preorder [NH84]. The stuck free conformance relation and the session type subtyping relation [GH99] are also more demanding than our subcontract relation. For instance $\mathbf{0}$ is not related with a in [FHRR04] whilst $\mathbf{0} \preceq a$.

It is worth noticing that must testing is preserved by any CCS context without $+$, and stuck free conformance is preserved by all CCS contexts thus allowing modular refinement. This is not true for $\preceq$. For instance $a \preceq a + b$ so one might think that a service with contract a can be replaced by a service with contract $a + b$ in any context. However, the context $C = \overline{b} \mid b.\overline{a} \mid [\,]$ distinguishes the two services ($a + b$ can get stuck while a cannot). The point is that the context C, representing a client, does not comply with a, since it performs the actions b and $\overline{b}$ which are not allowed by the contract a.

4.3 The finite contract language

The syntax of contracts uses an infinite set of *names* $\mathcal{N}$ ranged over by a, b, c, $\ldots$, and a disjoint set of *co-names* $\overline{\mathcal{N}}$ ranged over by $\overline{a}$, $\overline{b}$, $\overline{c}$, $\ldots$. We let $\overline{\overline{a}} = a$. Contracts σ are defined by the grammar in Table 4.1. Contracts are abstract definitions of conversation protocols between communicating parties. The contract $\mathbf{0}$ defines the empty conversation; the input prefix $a.\sigma$ defines a conversation protocol whose initial activity is to accept a message on the name a and continuing as σ; the output prefix $\overline{a}.\sigma$ defines a conversation protocol whose initial activity is to send a message to the name a and continuing as σ. Contracts

Table 4.1: The finite contract language.

σ ::=		**contracts**
	$\mathbf{0}$	(void)
	$a.\sigma$	(input prefix)
	$\overline{a}.\sigma$	(output prefix)
	$\sigma + \sigma$	(external choice)
	$\sigma \oplus \sigma$	(internal choice)

$\sigma + \sigma'$ and $\sigma \oplus \sigma'$ define conversation protocols that follow either the conversation σ or σ'; in the former ones the choice is left to the remote party, in the latter ones the choice being made locally. For example, $\texttt{Login}.(\overline{\texttt{Continue}} + \overline{\texttt{End}})$ describes the conversation protocol of a service that is ready to accept `Login`s and will `Continue` or `End` the conversation according to client's request. This contract is different from $\texttt{Login}.(\overline{\texttt{Continue}} \oplus \overline{\texttt{End}})$ where the decision whether to continue or to end is taken by the service.

In what follows, the trailing $\mathbf{0}$ is always omitted, α is used to range over names and co-names, and $\sum_{i\in 1..n} \sigma_i$ and $\bigoplus_{i\in 1..n} \sigma_i$ abbreviate $\sigma_1 + \cdots + \sigma_n$ and $\sigma_1 \oplus \cdots \oplus \sigma_n$, respectively.

4.3.1 Subcontract relation and dual contracts

Contracts retain an obvious compatibility relation that relates the conversation protocols of two communicating parties: a contract σ of a party *complies with* σ' of another party if the corresponding protocols match when they interact. Such a definition of subcontract would require the notions of communicating party, which is a process, and of contract exposed by it. We partially explore this direction in Section 4.5; here we give a direct definition by sticking to a structured operational semantics style. We begin by defining two notions that are preliminary to compliance: *subcontract* and *dual contract*.

Let $\sigma \downarrow \alpha$ be the least predicate on contracts such that

$$\begin{array}{ll} \alpha.\sigma \downarrow \alpha & \\ \sigma \oplus \sigma' \downarrow \alpha & \text{if } \sigma \downarrow \alpha \text{ or } \sigma' \downarrow \alpha \\ \sigma + \sigma' \downarrow \alpha & \text{if } \sigma \downarrow \alpha \text{ or } \sigma' \downarrow \alpha \end{array}$$

We write $\sigma \not\downarrow \alpha$ if $\sigma \downarrow \alpha$ is false. For instance $a \oplus b \not\downarrow c$.

Definition 4.1 (Transition $\stackrel{\alpha}{\longmapsto}$) *The* transition relation *of contracts, noted* $\stackrel{\alpha}{\longmapsto}$*, is the least relation satisfying the rules:*

$$\alpha.\sigma \stackrel{\alpha}{\longmapsto} \sigma$$

$$\frac{\sigma_1 \stackrel{\alpha}{\longmapsto} \sigma_1' \quad \sigma_2 \stackrel{\alpha}{\longmapsto} \sigma_2'}{\sigma_1 + \sigma_2 \stackrel{\alpha}{\longmapsto} \sigma_1' \oplus \sigma_2'} \qquad \frac{\sigma_1 \stackrel{\alpha}{\longmapsto} \sigma_1' \quad \sigma_2 \not\downarrow \alpha}{\sigma_1 + \sigma_2 \stackrel{\alpha}{\longmapsto} \sigma_1'}$$

$$\frac{\sigma_1 \stackrel{\alpha}{\longmapsto} \sigma_1' \quad \sigma_2 \stackrel{\alpha}{\longmapsto} \sigma_2'}{\sigma_1 \oplus \sigma_2 \stackrel{\alpha}{\longmapsto} \sigma_1' \oplus \sigma_2'} \qquad \frac{\sigma_1 \stackrel{\alpha}{\longmapsto} \sigma_1' \quad \sigma_2 \not\downarrow \alpha}{\sigma_1 \oplus \sigma_2 \stackrel{\alpha}{\longmapsto} \sigma_1'}$$

and closed under mirror cases for external and internal choices.

The relation $\stackrel{\alpha}{\longmapsto}$ is different from standard transition relations for CCS processes [Mil82]. For example, there is always at most one contract σ' such that $\sigma \stackrel{\alpha}{\longmapsto} \sigma'$, while this is not the case in CCS (the process $a.b + a.c$ has two different a-successor states: b and c). This mismatch is due to the fact that contract transitions define the evolution of conversation protocols *from the perspective of the communicating parties.* Thus $a.b + a.c \stackrel{a}{\longmapsto} b \oplus c$ because, once the activity a has been done, the communicating party is not aware of which conversation path has been chosen. On the contrary, CCS transitions define the evolution of processes *from the perspective of the process itself.*

Proposition 4.1 $\stackrel{\alpha}{\longmapsto}$ *is deterministic (for every contract σ, there is always at most one contract σ' such that $\sigma \stackrel{\alpha}{\longmapsto} \sigma'$).*

We write $\sigma(\alpha)$ for the unique continuation of σ after α, that is the contract σ' such that $\sigma \stackrel{\alpha}{\longmapsto} \sigma'$. We also define the set of traces of a contract σ, written $\texttt{traces}(\sigma)$, as $\texttt{traces}(\sigma) \stackrel{\text{def}}{=} \{\alpha_1 \cdots \alpha_n : \sigma \stackrel{\alpha_1}{\longmapsto} \cdots \stackrel{\alpha_n}{\longmapsto}\}$. It is easy to see that $\texttt{traces}(\sigma) \subseteq \texttt{traces}(\sigma')$ implies $\texttt{traces}(\sigma(\alpha)) \subseteq \texttt{traces}(\sigma'(\alpha))$.

Definition 4.2 (Ready sets) *Let* R *range over finite sets of names and co-names, called* ready sets*.* $\sigma \searrow \mathrm{R}$ *is the least relation such that:*

$$\begin{array}{ll}
\mathbf{0} \searrow \emptyset & \\
\alpha.\sigma \searrow \{\alpha\} & \\
(\sigma + \sigma') \searrow \mathrm{R} \cup \mathrm{R}' & \textit{if } \sigma \searrow \mathrm{R} \textit{ and } \sigma' \searrow \mathrm{R}' \\
(\sigma \oplus \sigma') \searrow \mathrm{R} & \textit{if either } \sigma \searrow \mathrm{R} \textit{ or } \sigma' \searrow \mathrm{R}
\end{array}$$

Ready sets define the sets of names and co-names where the contract may be waiting the interacting party. For instance the contract $a + b$ has ready set $\{a, b\}$ because describes a protocol where the party is waiting on both a and b. On the other hand the contract $a \oplus b$ has two ready sets $\{a\}$ and $\{b\}$ because the party can wait on one of the two names. Finally, the contract $(a \oplus b) + c$ has two ready sets $\{a, c\}$ and $\{b, c\}$ because either $(a \oplus b) + c$ evolves in $a + c$ or $(a \oplus b) + c$ evolves in $b + c$. It is worth to notice that the definition of ready sets and the definition of transition induce on contract a Moore automaton structure where the outputs are the set of ready sets and the transition function is given by $\longmapsto$. This formulation has been used by Boreale and Gadducci in [BG06].

Definition 4.3 (Subcontracts) *A* subcontract relation $\mathcal{S}$ *is a relation on contracts such that* $\sigma_1 \mathcal{S} \sigma_2$ *implies:*

1. *if* $\sigma_2 \searrow \mathrm{R}_2$ *then* $\sigma_1 \searrow \mathrm{R}_1$ *with* $\mathrm{R}_1 \subseteq \mathrm{R}_2$*,*
2. *if* $\sigma_1 \stackrel{\alpha}{\longmapsto} \sigma_1'$ *and* $\sigma_2 \stackrel{\alpha}{\longmapsto} \sigma_2'$ *then* $\sigma_1' \preceq \sigma_2'$*.*

Let $\preceq$ *be the largest subschema relation. Let* $\sigma_1 \simeq \sigma_2$*, called* contract equivalence*, if both* $\sigma_1 \preceq \sigma_2$ *and* $\sigma_2 \preceq \sigma_1$*.*

The relation $\sigma \preceq \sigma'$ verifies whether the external non-determinism of σ' is greater than the external non-determinism of σ and that this holds for every α-successor of σ and σ', *provided both have such successors*. For example $a.(b \oplus c) \simeq a.b + a.c \simeq a.b \oplus a.c$ and $a.b \oplus b \preceq b$ and $b \preceq b + a.c$. It is worth to remark that $\preceq$ is not transitive: the last two relations *do not entail* $a.b \oplus b \preceq b + a.c$, which is false. This transitivity failure is not very problematic because σ and σ' are intended to play different roles in $\sigma \preceq \sigma'$, as detailed by the compliance relation. However, transitivity of $\preceq$ holds under lightweight conditions.

Lemma 4.1 *If* $\sigma_1 \preceq \sigma_2$ *and* $\sigma_2 \preceq \sigma_3$ *and either (a)* $\mathtt{traces}(\sigma_1) \subseteq \mathtt{traces}(\sigma_2)$ *or (b)* $\mathtt{traces}(\sigma_3) \subseteq \mathtt{traces}(\sigma_2)$ *then* $\sigma_1 \preceq \sigma_3$*.*

Proof: Let $\mathcal{S}_1$ and $\mathcal{S}_2$ the two subcontract relations such that $(\sigma_1, \sigma_2) \in \mathcal{S}_1$ and $(\sigma_2, \sigma_3) \in \mathcal{S}_2$. Let $\mathcal{S}_3$ be the least relation such that $(\sigma_1, \sigma_3) \in \mathcal{S}_3$ if

– there exists σ_2 such that $(\sigma_1, \sigma_2) \in \mathcal{S}_1$ and $(\sigma_2, \sigma_3) \in \mathcal{S}_2$ and either:

$$\mathtt{traces}(\sigma_1') \subseteq \mathtt{traces}(\sigma_2') \quad \text{or} \quad \mathtt{traces}(\sigma_3') \subseteq \mathtt{traces}(\sigma_2')$$

We prove that $\mathcal{S}_3$ is a subcontract relation. As regards (1) we have to show that $\sigma_3 \searrow \mathrm{R}_3$ implies $\sigma_1 \searrow \mathrm{R}_1$ with $\mathrm{R}_1 \subseteq \mathrm{R}_3$. By $(\sigma_2, \sigma_3) \in \mathcal{S}_2$ there exists R_2 such that $\sigma_2 \searrow \mathrm{R}_2$ and $\mathrm{R}_2 \subseteq \mathrm{R}_3$. By $(\sigma_1, \sigma_2) \in \mathcal{S}_1$ there exists R_1 such that $\sigma_1 \searrow \mathrm{R}_1$ and $\mathrm{R}_1 \subseteq \mathrm{R}_2$. The item follows by transitivity of $\subseteq$. As regards (2) we have to show that $\sigma_1 \stackrel{\alpha}{\longmapsto} \sigma_1'$ and $\sigma_3 \stackrel{\alpha}{\longmapsto} \sigma_3'$ implies $(\sigma_1', \sigma_3') \in \mathcal{S}_3$. By either $\mathtt{traces}(\sigma_1) \subseteq \mathtt{traces}(\sigma_2)$ or $\mathtt{traces}(\sigma_3) \subseteq \mathtt{traces}(\sigma_2)$ we obtain $\sigma_2 \stackrel{\alpha}{\longmapsto} \sigma_2'$ with $(\sigma_1', \sigma_2') \in \mathcal{S}_1$ and $(\sigma_2', \sigma_3') \in \mathcal{S}_2$. Then we conclude $(\sigma_1', \sigma_3') \in \mathcal{S}_3$. $\square$

Proposition 4.2 $\sigma \oplus \sigma' \preceq \sigma$.

Proof: Let $\mathcal{S} = \{(\sigma \oplus \sigma', \sigma),\ (\sigma, \sigma) : \sigma,\ \sigma' \text{ are contracts}\}$. We show that $\mathcal{S}$ is a subcontract relation. Pairs (σ, σ) are immediate. As regards as pairs $(\sigma \oplus \sigma', \sigma)$, by definition of ready set $\sigma \searrow \mathrm{R}$ and $\sigma \oplus \sigma' \searrow \mathrm{R}$. Thus item 1 of $\preceq$ holds. Moreover, $\sigma \stackrel{\alpha}{\longmapsto} \sigma(\alpha)$ and $\sigma \oplus \sigma' \stackrel{\alpha}{\longmapsto} \sigma''$ implies either $\sigma'' = \sigma(\alpha)$ or $\sigma'' = \sigma(\alpha) \oplus \sigma'''$ for some σ'''. In both cases $(\sigma'', \sigma(\alpha)) \in \mathcal{S}$. $\square$

Proposition 4.3 *If* $\sigma \preceq \sigma' \oplus \sigma''$ *then* $\sigma \preceq \sigma'$ *and* $\sigma \preceq \sigma''$.

Proof: By the hypothesis, there is a subcontract relation $\mathcal{S}$ such that $(\sigma, \sigma' \oplus \sigma'') \in \mathcal{S}$. Let

$$\mathcal{R} = \{(\sigma, \sigma'), (\sigma, \sigma'') : (\sigma, \sigma' \oplus \sigma'') \in \mathcal{S}\} \cup \mathcal{S}.$$

We show that $\mathcal{R}$ is a subcontract relation. Since $\mathcal{S}$ is a subcontract relation we have:

$$\sigma' \oplus \sigma'' \searrow \mathrm{R} \quad \Rightarrow \quad \sigma \searrow \mathrm{R}' \subseteq \mathrm{R} \tag{4.1}$$

$$\sigma \stackrel{\alpha}{\longmapsto} \sigma(\alpha) \text{ and } \sigma' \oplus \sigma'' \stackrel{\alpha}{\longmapsto} (\sigma' \oplus \sigma'')(\alpha) \quad \Rightarrow \quad \sigma(\alpha) \mathcal{S} (\sigma' \oplus \sigma'')(\alpha) \tag{4.2}$$

We demonstrate the items.

- As regards ready sets we immediately conclude by (4.1)
- As regards transitions we discuss $\sigma' \stackrel{\alpha}{\longmapsto}$ (in case of $\sigma'' \stackrel{\alpha}{\longmapsto}$ we proceed in the same way). We have to show that $\sigma' \stackrel{\alpha}{\longmapsto} \sigma'(\alpha)$ and $\sigma \stackrel{\alpha}{\longmapsto} \sigma(\alpha)$ implies $(\sigma(\alpha), \sigma'(\alpha)) \in \mathcal{R}$. We have two cases: (a) $\sigma'' \not\downarrow \alpha$, and (b) $\sigma'' \downarrow \alpha$. In case (a), $\sigma' \oplus \sigma'' \stackrel{\alpha}{\longmapsto} \sigma'(\alpha)$ and we conclude $(\sigma(\alpha), \sigma'(\alpha)) \in \mathcal{R}$ by (4.2) and by $\mathcal{S} \subseteq \mathcal{R}$. In case (b), $\sigma' \oplus \sigma'' \stackrel{\alpha}{\longmapsto} \sigma'(\alpha) \oplus \sigma''(\alpha)$ and, by definition of $\mathcal{R}$, $(\sigma(\alpha), \sigma'(\alpha)) \in \mathcal{R}$, $(\sigma(\alpha), \sigma''(\alpha)) \in \mathcal{R}$. Thus we conclude. $\square$

The relation $\preceq$ is incomparable with may testing semantics [Hen88]: we have $a \oplus \mathbf{0} \preceq b$, while these two processes are unrelated by may testing; conversely, $a \oplus b$ and $a + b$ are may-testing equivalent, while $a + b \not\preceq a \oplus b$. The relation $\preceq$ is less discriminating than must testing semantics [Hen88]: a and $a + b$ are unrelated in must testing while $a \preceq a + b$.

The notion of *dual contract* is used to revert the capabilities of conversation protocols. Informally, the dual contract is obtained by reverting actions with co-actions, $+$ with $\oplus$, and conversely. For example the dual contract of $a \oplus \overline{b}$ is $\overline{a} + b$. However, this naïve transformation is fallible because in the contract language some external choices are actually internal choices in disguise. For example, $a.b + a.c \simeq a.(b \oplus c)$ but their naively constructed dual contracts are respectively $\overline{a}.\overline{b} \oplus \overline{a}.\overline{c}$ and $\overline{a}.(\overline{b} + \overline{c})$, and they tell very different things. In the first one, the communicating party cannot decide which action to perform after $\overline{a}$, whereas this possibility is granted in the second one. To avoid such misbehavior, we define dual contracts on contracts in normal form. We use the same forms introduced in [Hen88]. Let the *normed contract* of σ, noted $\mathtt{nc}(\sigma)$, be

$$\mathtt{nc}(\sigma) \stackrel{\text{def}}{=} \bigoplus_{\sigma \searrow \mathrm{R}} \sum_{\alpha \in \mathrm{R}} \alpha.\mathtt{nc}(\sigma(\alpha)) \ .$$

For example

$$\begin{array}{rll} \mathtt{nc}((a.b \oplus b.c) + (a.b.d \oplus c.b)) = & & a.b.(\mathbf{0} \oplus d) \\ & \oplus & (a.b.(\mathbf{0} \oplus d) + c.b) \\ & \oplus & (a.b.(\mathbf{0} \oplus d) + b.c) \\ & \oplus & (b.c + c.b) \end{array}$$

The following proposition is an immediate consequece of the definition of normed contract and is a preliminar for Lemma 4.2 which shows that a contract is subcontract equivalent to its normal form.

Proposition 4.4

1. $\sigma \searrow \mathrm{R}$ *if and only if* $\mathtt{nc}(\sigma) \searrow \mathrm{R}$
2. $\sigma \stackrel{\alpha}{\longmapsto} \sigma(\alpha)$ *if and only if* $\mathtt{nc}(\sigma) \stackrel{\alpha}{\longmapsto} \mathtt{nc}(\sigma(\alpha))$

Lemma 4.2 $\sigma \simeq \mathtt{nc}(\sigma)$ *and* $\mathtt{traces}(\sigma) = \mathtt{traces}(\mathtt{nc}(\sigma))$.

Proof: The subcontract relation equating σ and $\mathtt{nc}(\sigma)$ is $\mathcal{R} = \{(\sigma, \mathtt{nc}(\sigma)),\ (\mathtt{nc}(\sigma), \sigma) : \sigma \text{ is a contract}\}$. The fact that $\mathcal{R}$ is a subcontract relation follows immediately by Proposition 4.4. $\square$

Notice that $\preceq$ is not a pre-congruence. For example $a.b \preceq a.b + b.a$ but $a.b + b.c \not\preceq a.b + b.a + b.c$ because $a.b + b.c \overset{b}{\longmapsto} c$ and $a.b + b.a + b.c \overset{b}{\longmapsto} a \oplus c$ and $c \not\preceq a \oplus c$. Similarly $a \preceq a + b.a$ but $a \oplus b.c \not\preceq (a + b.a) \oplus b.c$ because $a + b.c \overset{b}{\longmapsto} c$ and $(a + b.a) \oplus b.c \overset{b}{\longmapsto} a \oplus c$ and $c \not\preceq a \oplus c$.

Now, to obtain the dual of a contract σ we first normalize it and then exchange actions with co-actions, $\oplus$ with $+$, and vice-versa. All these transformations are summarized by the following definition.

Definition 4.4 (Dual contracts) *The* dual contract *of σ, noted $\overline{\sigma}$, is defined as*

$$\overline{\sigma} \overset{\text{def}}{=} \textstyle\sum_{\sigma \searrow \mathrm{R}} \bigoplus_{\alpha \in \mathrm{R}} \overline{\alpha}.\overline{\sigma(\alpha)}$$

where, by convention, we have $\bigoplus_{\sigma \in \emptyset} \sigma = \mathbf{0}$.

The following propositions hold

Proposition 4.5

1. $\overline{\sigma} \searrow \mathrm{R}$ *if and only if* $\{\mathrm{R}_1, \dots, \mathrm{R}_n\} = \{\mathrm{R}' : \sigma \searrow \mathrm{R}'\},\ \mathrm{R} = \bigcup_{i \in 1..n} \overline{\alpha_i}, \alpha_i \in \mathrm{R}_i$*;*
2. *if* $\overline{\sigma} \searrow \emptyset$ *then* $\sigma \searrow \mathrm{R}$ *implies* $\mathrm{R} = \emptyset$*;*
3. *if* $\sigma \overset{\alpha}{\longmapsto} \sigma(\alpha)$ *then* $\overline{\sigma} \overset{\overline{\alpha}}{\longmapsto} \overline{\sigma(\alpha)}$*.*

Every item is an immediate consequence of the definition of dual. In particular, item 2 follows by the fact that the dual contract is an external choice of internal choices. Thus, if we have $\sigma \searrow \emptyset$ then every $\bigoplus_{\alpha \in \mathrm{R}} \alpha.\sigma(\alpha) \searrow \emptyset$. This implies $\mathrm{R} = \emptyset$.

The dual operator is not contravariant with respect to $\preceq$. For example, $a \preceq a.b$, but $\overline{a.b} = \overline{a}.\overline{b} \not\preceq \overline{a}$. For similar reasons, contract compatibility is not preserved. For example, $\mathbf{0} \simeq \mathbf{0} \oplus a$ but $\overline{\mathbf{0}} = \mathbf{0} \not\simeq \mathbf{0} + \overline{a} = \overline{\mathbf{0} \oplus a}$. However a limited form of contravariance, which will result fundamental in the following, is satisfied by the dual operator.

Lemma 4.3 $\overline{\sigma} \preceq \overline{\sigma \oplus \sigma'}$.

Proof: Let

$$\mathcal{R} = \{(\overline{\sigma}, \overline{\sigma \oplus \sigma'}),\ (\overline{\sigma}, \overline{\sigma}) : \sigma,\ \sigma' \text{ are contracts}\}.$$

We show that $\mathcal{R}$ is a subcontract relation:

- As regards ready sets, by Proposition 4.5 applied to $\overline{\sigma \oplus \sigma'}$ and by definition of ready set of $\oplus$, $\overline{\sigma \oplus \sigma'} \searrow \mathrm{R}'$ implies $\{\mathrm{R}_1, \ldots, \mathrm{R}_n, \mathrm{S}_1, \ldots, \mathrm{S}_m\} = \{\mathrm{R} : \sigma \searrow \mathrm{R}\} \cup \{\mathrm{S} : \sigma' \searrow \mathrm{S}\}$, $\mathrm{R}' = \bigcup_{i \in \{1,\ldots,n\}} \overline{\alpha_i} \cup \bigcup_{j \in \{1,\ldots,m\}} \overline{\beta_j}$ with $\alpha_i \in \mathrm{R}_i, \beta_j \in \mathrm{S}_j$. By Proposition 4.5 applied to $\overline{\sigma}$, since $\overline{\sigma} \searrow \mathrm{R}''$ with $\{\mathrm{R}_1 \ldots \mathrm{R}_n\} = \{\mathrm{R} : \sigma \searrow \mathrm{R}\}$, $\mathrm{R}'' = \bigcup_{i \in 1..n} \overline{\alpha_i}, \alpha_i \in \mathrm{R}_i$. Thus we conclude $\mathrm{R}'' \subseteq \mathrm{R}'$.
- As regards transitions, by Proposition 4.5 we have $\overline{\sigma} \overset{\alpha}{\longmapsto} \overline{\sigma(\alpha)}$ and $\overline{\sigma \oplus \sigma'} \overset{\alpha}{\longmapsto} \overline{(\sigma \oplus \sigma')(\alpha)}$. We must show that $(\overline{\sigma(\alpha)}, \overline{(\sigma \oplus \sigma')(\alpha)}) \in \mathcal{R}$. We have two cases: $\sigma' \downarrow \alpha$ and $\sigma' \not\downarrow \alpha$. In the first case $\overline{(\sigma \oplus \sigma')(\alpha)} = \overline{\sigma(\alpha) \oplus \sigma'(\alpha)}$. In the latter case $\overline{(\sigma \oplus \sigma')(\alpha)} = \overline{\sigma(\alpha)}$. Then in both cases we conclude by definition of $\mathcal{R}$. $\square$

4.3.2 Contract compliance

Every preliminary notion has been set for the definition of contract compliance.

Definition 4.5 (Contract compliance) *A contract σ* complies with *σ', noted $\sigma \between_c \sigma'$, if and only if $\overline{\sigma} \preceq \sigma'$.*

The notion of contract compliance is meant to be used for querying a Web service repository. A client with contract σ will interact successfully with every service with contract σ' provided $\sigma \between_c \sigma'$. For example, consider a client whose conversation protocol states that it intends to choose whether to be notified either on a name a or on a name b. Its contract might be $a \oplus b$. Querying a repository for compliant services means returning every service whose conversation protocol is $\overline{a}+\overline{b}$, or $\overline{a}+\overline{b}+a$, or $\overline{a}.c+\overline{b}$, etc. The guarantee that we provide (see Section 4.5) is that, whatever service returned by the repository is chosen, the client will conclude his conversation. This asymmetry between the left hand side of $\preceq$ (and of $\between_c$) and the right hand side is the reason of the failure of transitivity. More precisely, in $a.b \oplus b \preceq b$ and in $b \preceq a.c + b$, we are guaranteeing the termination of

clients manifesting the two left hand sides contracts with respect to services manifesting the two right hand side contracts. This property is not transitive.

Another case is when clients query the repository with $\mathbf{0} \oplus \sigma$. Any such client must be able to complete immediately, because it is not possible to distinguish between a service that has decided to behave as $\mathbf{0}$ and one that has decided to behave as σ, but it is very slow in performing the next action.

4.3.3 Contract structural equivalence

Definition 4.6 *The contract structural equivalence is the least relation over contracts closed and under the rules in Table 4.2*

Table 4.2: Axioms for $=$.

$\sigma + \sigma = \sigma$	$\sigma \oplus \sigma = \sigma$
$\sigma + \sigma' = \sigma' + \sigma$	$\sigma \oplus \sigma' = \sigma' \oplus \sigma$
$\sigma + (\sigma' + \sigma'') = (\sigma + \sigma') + \sigma''$	$\sigma \oplus (\sigma' \oplus \sigma'') = (\sigma \oplus \sigma') \oplus \sigma''$
$\sigma + \mathbf{0} = \sigma$	
$\alpha.\sigma + \alpha.\sigma' = \alpha.(\sigma \oplus \sigma')$	$\alpha.\sigma \oplus \alpha.\sigma' = \alpha.(\sigma \oplus \sigma')$
$\sigma + (\sigma' \oplus \sigma'') = (\sigma + \sigma') \oplus (\sigma + \sigma'')$	$\sigma \oplus (\sigma' + \sigma'') = (\sigma \oplus \sigma') + (\sigma \oplus \sigma'')$

Proposition 4.6 *If* $\sigma = \sigma'$ *then* $\sigma \simeq \sigma'$*.*

Proof: Let $\mathcal{S} = \{(\sigma, \sigma') : \sigma = \sigma'\}$. We show that $\mathcal{S}$ is a subcontract relation. Let $(\sigma, \sigma') \in \mathcal{S}$, as regard s ready sets, every equivalence preserve the set of ready sets except the distributivity law $\sigma \oplus (\sigma' + \sigma'') = (\sigma \oplus \sigma') + (\sigma \oplus \sigma'')$. We show that in this case condition 1 of the subcontract relation holds. We have that:

- $\sigma \oplus (\sigma' + \sigma'') \searrow \mathrm{R}$ implies either: $\sigma \searrow \mathrm{R}$ or $\sigma' \searrow \mathrm{R}'$, $\sigma'' \searrow \mathrm{R}''$, and $\mathrm{R} = \mathrm{R}' \cup \mathrm{R}''$
- $(\sigma \oplus \sigma') + (\sigma \oplus \sigma'') \searrow \mathrm{R}$ implies either: $\sigma \searrow \mathrm{R}$ or $\sigma \searrow \mathrm{R}'$, $\sigma'' \searrow \mathrm{R}''$, and $\mathrm{R} = \mathrm{R}' \cup \mathrm{R}''$ or $\sigma' \searrow \mathrm{R}'$, $\sigma \searrow \mathrm{R}''$, and $\mathrm{R} = \mathrm{R}' \cup \mathrm{R}''$ or $\sigma' \searrow \mathrm{R}'$, $\sigma'' \searrow \mathrm{R}''$, and $\mathrm{R} = \mathrm{R}' \cup \mathrm{R}''$.

Then it is easy to see that for every ready set of $\sigma \oplus (\sigma' + \sigma'')$, $(\sigma \oplus \sigma') + (\sigma \oplus \sigma'')$ owns a smaller ready set, and vice-versa. Thus we conclude. $\square$

4.4 Message exchange patterns in WSDL

The Web Service Description Language (WSDL) Version 1.1 [Gro] permits to describe and publish abstract and concrete descriptions of Web services. Such descriptions include the schema [Gro04b] of messages exchanged between client and server, the name and type of *operations* that the service exposes, as well as the locations (URLs) where the service can be contacted. In addition, it defines four interaction patterns determining the order and direction of exchanged messages. For instance, the *request-response* pattern is used to describe a synchronous operation where the client issues a request and subsequently receives a response from the service.

The second version of WSDL [BL06, CHL+06, CMRW06] allows users to agree on message exchange patterns (MEP) by specifying in the required `pattern` attribute of operation elements an absolute URI that identifies the MEP. It is important to notice that these URIs act as global identifiers (their content is not important) for MEPs, whose semantics is usually given in plain English.

In particular, WSDL2.0 [CHL+06] predefines four message exchange patterns (each pattern being uniquely identified by a different URI) for describing services where the interaction is initiated by clients (four further MEPs are provided for interactions initiated by servers). Let us shortly discuss how the informal plain English semantics of these patterns can be formally defined in our contract language. Consider the WSDL 2.0 fragment (without schemas)

```
< operation name = "o1" pattern = "In-Only" >
  < input Label = "In"/ >
< /operation >
< operation name = "o2" pattern = "Robust-In-Only" >
  < input Label = "In"/ >
  < outfault Label = "Fault"/ >
< /operation >
< operation name = "o3" pattern = "In-Out" >
  < input Label = "In"/ >
  < output Label = "Out"/ >
  < outfault Label = "Fault"/ >
< /operation >
```

```
< operation name = "o4" pattern = "In-Opt-Out" >
  < input Label = "In"/ >
  < output Label = "Out"/ >
  < outfault Label = "Fault"/ >
< /operation >
```

which defines four operations named o_1, o_2, o_3, and o_4. The first two operations are *asynchronous* by accepting only an incoming message labeled `In`. The last two operations are *synchronous* by accepting an incoming message labeled `In` and replying with a message labeled `Out`. In the o_2 operation a fault message can occur after the input. The o_3 operation always produces an output message (see `In-Out` in its `pattern` attribute), unless a fault occurs. In the o_4 operation the reply is optional, as stated by the `In-Opt-Out` exchange pattern attribute, and again it may fail with `Fault`.

We can encode the contract of the pattern of the o_1 operation in our contract language as

$$\mathtt{inOnly} = \mathtt{In}.\overline{\mathtt{End}}$$

that is an input action representing the client's request followed by a message $\overline{\mathtt{End}}$ that is sent from the service to notify the client that the interaction has completed.

The o_2 operation can be encoded as

$$\mathtt{robustInOnly} = \mathtt{In}.(\overline{\mathtt{End}} \oplus \overline{\mathtt{Fault}}.\overline{\mathtt{End}})$$

where after the client's request, the interaction may follow two paths, representing successful and faulty computations respectively. In the former case the end of the interaction is immediately signaled to the client. In the latter case a message `Fault` is sent to the client, followed by `End`. The use of the internal choice for combining the two paths states that it is the service that decides whether the interaction is successful or not. This means that a client compliant with this service can either stop after the request or it must be able to handle both the `End` and `Fault` messages: the omission of handling, say, `Fault` would result into an uncaught exception.

The need for an explicit `End` message to signal a terminated interaction is not immediately evident. In principle, the optional fault message could have been encoded as $\mathtt{In}.(\mathbf{0} \oplus \overline{\mathtt{Fault}})$. A client compliant with this service must be able to receive and handle the

`Fault` message, but it must also be able to complete the interaction without further communication from the service. The point is that the client cannot distinguish a completed interaction where the service has internally decided to behave like **0** from an interaction where the service has internally decided to behave like `Fault`, but it is taking a long time to respond. By providing an explicit `End` message signaling a completed interaction, the service tells the client not to wait for further messages. By this reasoning, the `End` message after `Fault` is not strictly necessary, but we write it for uniformity.

By similar arguments the contract of the o_3 operation can be encoded as

$$\mathtt{inOut} = \mathtt{In}.(\overline{\mathtt{Out}}.\overline{\mathtt{End}} \oplus \overline{\mathtt{Fault}}.\overline{\mathtt{End}})$$

and the contract of the o_4 operation as

$$\mathtt{inOptOut} = \mathtt{In}.(\overline{\mathtt{End}} \oplus \overline{\mathtt{Out}}.\overline{\mathtt{End}} \oplus \overline{\mathtt{Fault}}.\overline{\mathtt{End}})$$

It is worth noticing how these contracts are ordered according to our definition of $\preceq$. We have `inOptOut` $\preceq$ `robustInOnly` and `robustInOnly` $\preceq$ `inOnly`. Indeed, a client compliant with `inOptOut` must be able to complete immediately after the request, but it is also able to handle a `Out` message and a `Fault` message. The `robustInOnly` can only produce an `End` message or a `Fault` message, hence it is "more deterministic" than `inOptOut`. Similarly, `inOnly` is more deterministic than `robustInOnly` since it can only send an `End` message after the client's request. Finally, note that `inOptOut` $\preceq$ `inOut` also holds.

By combining contracts we can express the compound contract exported by the service as

$$\begin{aligned} & o_1.\mathtt{In}.\overline{\mathtt{End}} \\ + \; & o_2.\mathtt{In}.(\overline{\mathtt{End}} \oplus \overline{\mathtt{Fault}}.\overline{\mathtt{End}}) \\ + \; & o_3.\mathtt{In}.(\overline{\mathtt{Out}}.\overline{\mathtt{End}} \oplus \overline{\mathtt{Fault}}.\overline{\mathtt{End}}) \\ + \; & o_4.\mathtt{In}.(\overline{\mathtt{End}} \oplus \overline{\mathtt{Out}}.\overline{\mathtt{End}} \oplus \overline{\mathtt{Fault}}.\overline{\mathtt{End}}) \end{aligned}$$

where the external choice makes it explicit that the choice among o_1, o_2, o_3 and o_4 must be made by the client.

4.5 Process Compliance

Compliance relates a client process with a service process. A client is compliant with a service if the client terminates (i.e. it has no more interactions to perform) for every

possible interaction with the service. That is, compliance induces a *completion property* for the client but not for the service. In order to formalize compliance we define processes and their dynamics. Then we demonstrate that it is possible to associate a contract to a process such that (process) compliance follows by the compliance of the corresponding contracts.

In this chapter, processes are finite CCS terms. For the sake of simplicity we do not include choice and relabeling operators. We note that, omitting choice, there are contracts that cannot be associated to any process (i.e. $a + b$, $a \oplus b$). However we show hot to deal with choices in the next Chapter, where recursive CCS processes are also considered.

The transition relation is standard; therefore we omit comments.

Definition 4.7 *Finite* CCS *processes P are defined by the following grammar:*

$$P \ ::= \quad \mathbf{0} \quad | \quad a.P \quad | \quad \overline{a}.P \quad | \quad P \setminus a \quad | \quad P \mid P$$

Let μ range over $\mathcal{N} \cup \overline{\mathcal{N}} \cup \{\tau\}$. The transition relation *of processes, noted $\xrightarrow{\mu}$, is the least relation satisfying the rules:*

$$\text{(IN)}\quad a.P \xrightarrow{a} P \qquad \text{(OUT)}\quad \overline{a}.P \xrightarrow{\overline{a}} P \qquad \text{(RES)}\quad \frac{P \xrightarrow{\mu} Q \quad \mu \notin \{a, \overline{a}\}}{P \setminus a \xrightarrow{\mu} Q \setminus a}$$

$$\text{(PAR)}\quad \frac{P \xrightarrow{\mu} Q}{P \mid R \xrightarrow{\mu} Q \mid R} \qquad \text{(COM)}\quad \frac{P \xrightarrow{\alpha} P' \quad Q \xrightarrow{\overline{\alpha}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'}$$

The transitions of $P \mid Q$ have mirror cases that have been omitted.

We write $\stackrel{\tau}{\Longrightarrow}$ for $\xrightarrow{\tau}^{*}$ and $\stackrel{\alpha}{\Longrightarrow}$ for $\xrightarrow{\tau}^{*}\xrightarrow{\alpha}\xrightarrow{\tau}^{*}$.

The compliance of a client process with a service is defined as follows.

Definition 4.8 (Compliance) *Let $P \,\|\, Q \longrightarrow P' \,\|\, Q'$ be the least relation such that:*

- *if $P \xrightarrow{\tau} P'$ then $P \,\|\, Q \longrightarrow P' \,\|\, Q$;*
- *if $Q \xrightarrow{\tau} Q'$ then $P \,\|\, Q \longrightarrow P \,\|\, Q'$;*
- *if $P \xrightarrow{\alpha} P'$ and $Q \xrightarrow{\overline{\alpha}} Q'$ then $P \,\|\, Q \longrightarrow P' \,\|\, Q'$.*

Let $P \between_p Q$, *read* P complies with Q, *if one of the following holds:*

1. $P \stackrel{\mu}{\not\longrightarrow}$, *or*

2. $P \parallel Q \longrightarrow P' \parallel Q'$ *and* $P' \between_p Q'$.

Process compliance has been noted in the same way as contract compliance in Section 4.3. This abuse is justified because the two notions are strongly related, as we will prove shortly.

Processes expose (principal) contracts. This is defined by an inference system that uses two auxiliary operators over contracts:

1. $\sigma \setminus a$ is defined by induction on the structure of σ:

$$\begin{array}{rcl} \mathbf{0} \setminus a & = & \mathbf{0} \\ (\alpha.\sigma) \setminus a & = & \begin{cases} \mathbf{0} & \text{if } \alpha \in \{a, \overline{a}\} \\ \alpha.(\sigma \setminus a) & \text{otherwise} \end{cases} \\ (\sigma + \sigma') \setminus a & = & \sigma \setminus a + \sigma' \setminus a \\ (\sigma \oplus \sigma') \setminus a & = & \sigma \setminus a \oplus \sigma' \setminus a \end{array}$$

2. The operator "|" is commutative with $\mathbf{0}$ as identity, such that $\sigma \mid (\sigma' \oplus \sigma'') = (\sigma \mid \sigma') \oplus (\sigma \mid \sigma'')$, and $\sigma \mid (\sigma' + (\sigma'' \oplus \sigma''')) = \sigma \mid ((\sigma' + \sigma'') \oplus (\sigma' + \sigma'''))$. This allows us to define $\sigma \mid \sigma'$ when σ and σ' are external choices of prefixes. Our definition corresponds to the *expansion law* in [NH87]. Let $\sigma = \sum_{i \in I} \alpha_i.\sigma_i$ and $\sigma' = \sum_{j \in J} \alpha'_j.\sigma'_j$, then

$$\sigma \mid \sigma' \stackrel{\text{def}}{=} \begin{cases} \sum_{i \in I} \alpha_i.(\sigma_i \mid \sigma') \; + \; \sum_{j \in J} \alpha'_j.(\sigma \mid \sigma'_j) \\ \qquad\qquad\qquad\qquad \text{if } \alpha_i \neq \overline{\alpha'_j} \text{ for every } i \in I, \; j \in J \\ \\ \Big(\sum_{i \in I} \alpha_i.(\sigma_i \mid \sigma') \; + \; \sum_{j \in J} \alpha'_j.(\sigma \mid \sigma'_j) \; + \; \bigoplus_{\alpha_i = \overline{\alpha'_j}} (\sigma_i \mid \sigma'_j) \Big) \\ \quad \oplus \bigoplus_{\alpha_i = \overline{\alpha'_j}} (\sigma_i \mid \sigma'_j) \qquad\qquad\qquad\qquad \text{otherwise} \end{cases}$$

Proposition 4.7

1. $\sigma \preceq \sigma'$ *implies* $\sigma \setminus a \preceq \sigma' \setminus a$*;*

2. $\overline{\sigma} \preceq \overline{\sigma'}$ *implies* $\overline{\sigma \setminus a} \preceq \overline{\sigma' \setminus a}$.

Proof: As regards 1, let

$$\mathcal{S} = \{(\sigma \setminus a, \sigma' \setminus a) : \sigma \preceq \sigma'\}.$$

We show that $\mathcal{S}$ is a subcontract relation. By the hypothesis $\sigma \preceq \sigma'$, $\sigma' \searrow \mathrm{R}'$ and $\sigma \searrow \mathrm{R}$ and $\mathrm{R} \subseteq \mathrm{R}'$. By definition of "$\setminus$", $\sigma' \searrow \mathrm{R}' \setminus a$ and $\sigma \searrow \mathrm{R} \setminus a$. Hence we conclude $\mathrm{R} \setminus a \subseteq \mathrm{R}' \setminus a$. By the hypothesis $\sigma \preceq \sigma'$, we also have $\sigma(\alpha) \preceq \sigma'(\alpha)$. By definition of "$\setminus$", $\sigma \setminus a \stackrel{\alpha}{\longmapsto} \sigma(\alpha) \setminus a$ and $\sigma' \stackrel{\alpha}{\longmapsto} \sigma'(\alpha) \setminus a$. Hence we conclude $(\sigma(\alpha) \setminus a, \sigma'(\alpha) \setminus a) \in \mathcal{S}$.

As regards 2, let

$$\mathcal{S} = \{(\overline{\sigma \setminus a}, \overline{\sigma' \setminus a}) : \overline{\sigma} \preceq \overline{\sigma'}\}.$$

We show that $\mathcal{S}$ is a subcontract relation. By definition of dual contract we have

$$\begin{array}{rcl} \overline{\sigma \setminus a} & = & \sum_{\sigma \searrow \mathrm{R} \setminus \{a\}} \bigoplus_{\alpha \in \mathrm{R} \setminus \{a\}} \overline{\alpha}.\overline{\sigma(\alpha) \setminus a} \\ \overline{\sigma' \setminus a} & = & \sum_{\sigma' \searrow \mathrm{R} \setminus \{a\}} \bigoplus_{\alpha \in \mathrm{R} \setminus \{a\}} \overline{\alpha}.\overline{\sigma'(\alpha) \setminus a} \end{array}$$

Thus, condition 1 of $\preceq$, is immediate. To show condition 2 of $\preceq$ we must prove that $(\overline{\sigma(\alpha) \setminus a}, \overline{\sigma'(\alpha) \setminus a}) \in \mathcal{S}$. This follows by $\sigma(\alpha) \preceq \sigma'(\alpha)$. □

Definition 4.9 *Let $P \vdash \sigma$ be the least relation such that*

$$\mathbf{0} \vdash \mathbf{0} \qquad \frac{P \vdash \sigma}{a.P \vdash a.\sigma} \qquad \frac{P \vdash \sigma}{\overline{a}.P \vdash \overline{a}.\sigma} \qquad \frac{P \vdash \sigma}{P \setminus a \vdash \sigma \setminus a} \qquad \frac{P \vdash \sigma \quad Q \vdash \sigma'}{P \mid Q \vdash \sigma \mid \sigma'}$$

For instance we have $a \mid b \vdash a.b{+}b.a$ and $a \mid \overline{a} \vdash (a{+}\overline{a}{\oplus}\mathbf{0}){\oplus}\mathbf{0}$. As anticipated, compliance of processes may be inferred from compliance of the corresponding contracts. This property, formalized in Theorem 4.1, requires few preliminary statements. However, we note that the converse does not hold: if two processes comply then the correspondent contract may be not compliant. For instance $a \mid \overline{a} \between_p \mathbf{0}$ but $(a{+}\overline{a}{\oplus}\mathbf{0}){\oplus}\mathbf{0} \not\between_c \mathbf{0}$ ($\overline{(a + \overline{a} \oplus \mathbf{0}) \oplus \mathbf{0}} = \overline{a}{\oplus}a \not\preceq \mathbf{0}$).

Lemma 4.4 *Let $P \vdash \sigma$, $P \stackrel{\mu}{\longrightarrow} P'$, and $P' \vdash \sigma'$*

(a) *if $\mu = \tau$ then $\sigma \preceq \sigma'$, $\overline{\sigma'} \preceq \overline{\sigma}$, and $\mathtt{traces}(\sigma') \subseteq \mathtt{traces}(\sigma)$;*

(b) *if $\mu = \alpha$ then $\sigma(\alpha) \preceq \sigma'$, $\overline{\sigma'} \preceq \overline{\sigma(\alpha)}$, and $\mathtt{traces}(\sigma') \subseteq \mathtt{traces}(\sigma(\alpha))$.*

Proof: We proceed by induction on the derivation of $P \stackrel{\mu}{\longrightarrow} P'$.

The base case corresponds to the application of either (IN) or (OUT). Since P has the form $\alpha.P'$ we have $\sigma(\alpha) = \sigma'$. Therefore we conclude $\sigma(\alpha) \preceq \sigma'$, $\overline{\sigma'} \preceq \overline{\sigma(\alpha)}$, and $\mathtt{traces}(\sigma') = \mathtt{traces}(\sigma(\alpha))$.

In the inductive case there are several sub-cases corresponding to the last rule that has been applied.

- (COM) implies $P = Q \mid R$ with $Q \xrightarrow{\alpha} Q'$ and $R \xrightarrow{\overline{\alpha}} R'$. Let $Q \vdash \sigma_1$, $Q' \vdash \sigma'_1$, $R \vdash \sigma_2$, and $R' \vdash \sigma'_2$. By definition of "|", we have $\sigma_1 \mid \sigma_2 = \bigoplus_{i \in I} \sigma''_i$ with $\sigma''_j = \sigma'_1 \mid \sigma'_2$ for some $j \in I$. Hence $\sigma_1 \mid \sigma_2 \preceq \sigma'_1 \mid \sigma'_2$ follows by definition of $\preceq$ and $\overline{\sigma'_1 \mid \sigma'_2} \preceq \overline{\sigma_1 \mid \sigma_2}$ follows by Lemma 4.3. It remains to show $\texttt{traces}(\sigma'_1 \mid \sigma'_2) \subseteq \texttt{traces}(\sigma_1 \mid \sigma_2)$. This is a straightforward consequence of the definition of "|" and $\texttt{traces}(\cdot)$.

- (RES) implies $P = Q \setminus a$, $Q \xrightarrow{\mu} Q'$. Let $\sigma = \sigma_1 \setminus a$, $Q \vdash \sigma_1$, and $Q' \vdash \sigma'_1$.
  - If $\mu = \tau$ the lemma follows by Proposition 4.7 and by the hypotheses we conclude $\sigma_1 \preceq \sigma'_1$ and $\overline{\sigma'_1} \preceq \overline{\sigma_1}$.
  - If $\mu = \alpha$ the lemma follows by Proposition 4.7 and by the hypotheses we conclude $\sigma_1(\alpha) \preceq \sigma'_1$ and $\overline{\sigma'_1} \preceq \overline{\sigma_1(\alpha)}$.

- (PAR) implies $P = Q \mid R$ with $Q \xrightarrow{\mu} Q'$ and $Q \vdash \sigma_1$, $R \vdash \sigma_2$, and $Q' \vdash \sigma'_1$.
  - If $\mu = \tau$, by definition of "|", we have $\sigma_1 = \bigoplus_{i \in I} \sigma''_i$ with $\sigma''_j = \sigma'_1$ for some $j \in I$. Then $\sigma_1 \mid \sigma_2 = (\bigoplus_{i \in I} \sigma''_i) \mid \sigma_2 = \bigoplus_{i \in I} (\sigma''_i \mid \sigma_2)$ and $\sigma_1 \mid \sigma_2 \preceq \sigma'_1 \mid \sigma_2$ follows by Proposition 4.2 while $\overline{\sigma'_1 \mid \sigma_2} \preceq \overline{\sigma_1 \mid \sigma_2}$ follows by Lemma 4.3. By definition of $\texttt{traces}(\cdot)$ we also conclude that $\texttt{traces}(\sigma'_1 \mid \sigma_2) \subseteq \texttt{traces}(\sigma_1 \mid \sigma_2)$.
  - If $\mu = \alpha$, by the inductive hypothesis we have $\sigma_1(\alpha) \preceq \sigma'_1$ and $\overline{\sigma'_1} \preceq \overline{\sigma(\alpha)}$. Since $Q \xrightarrow{\alpha} Q'$, by definition of "|" we have that $\sigma_1 \mid \sigma_2 = \rho_1 \oplus (\rho_2 + \alpha.(\sigma'_1 \mid \sigma_2) + \rho_3) \oplus \rho_4$. Hence $(\sigma_1 \mid \sigma_2)(\alpha) = \bigoplus_{i \in I} \sigma_i$ with $\sigma_i = (\sigma'_1 \mid \sigma_2)$ for some $i \in I$. Then $(\sigma_1 \mid \sigma_2)(\alpha) \preceq \sigma'_1 \mid \sigma_2$ follows by definition of $\preceq$ and $\overline{\sigma'_1 \mid \sigma_2} \preceq \overline{(\sigma_1 \mid \sigma_2)(\alpha)}$ by Lemma 4.3. By definition of $\texttt{traces}(\cdot)$ we also conclude that $\texttt{traces}(\sigma'_1 \mid \sigma_2) \subseteq \texttt{traces}((\sigma_1 \mid \sigma_2)(\alpha))$. $\square$

Theorem 4.1 *If $P \vdash \sigma$, $Q \vdash \sigma'$, and $\sigma \between_c \sigma'$ then $P \between_p Q$.*

Proof: A maximal computation of the system $P \parallel Q$ is a sequence of systems $P_1 \parallel Q_1, \ldots, P_n \parallel Q_n$ such that $P_1 = P$, $Q_1 = Q$, for every $i = \{1, \ldots, n-1\}$ we have $P_i \parallel Q_i \longrightarrow P_{i+1} \parallel Q_{i+1}$, and $P_n \parallel Q_n \not\longrightarrow$. The proof is by induction on n.

If $n = 0$, then $P \parallel Q \not\longrightarrow$. We have two possibilities: if $P \stackrel{\mu}{\not\longrightarrow}$ then by definition $P \between_p Q$. So let us suppose, by contradiction, that whenever $P \xrightarrow{\alpha}$ we have $Q \stackrel{\overline{\alpha}}{\not\longrightarrow}$. Since $P \vdash \sigma$

and $Q \vdash \sigma'$ this means that for any ready set R of σ there is no ready set S of σ' such that $\overline{\mathrm{R}} \cap \mathrm{S} \neq \emptyset$. From $P \xrightarrow{\alpha}$ and $P \vdash \sigma$ we know that $\sigma \searrow \mathrm{R}$ and $\alpha \in \mathrm{R}$ for some ready set R. That is, σ has at least one nonempty ready set. Thus, from the definition of $\overline{\sigma}$, we know that *every* ready set of $\overline{\sigma}$ is not empty. By definition of contract compliance we know that $\overline{\sigma} \preceq \sigma'$ and from the definition of $\preceq$ we have that any ready set S of σ' shares at least an action with $\overline{\mathrm{R}}$ for some ready set R of σ, which is absurd.

If $n > 0$, assume that the theorem is true for any computation of length $n - 1$. We have three cases:

($P \xrightarrow{\tau} P'$) Assume $P' \vdash \sigma''$, then from Lemma 4.4(a) we know that $\overline{\sigma''} \preceq \overline{\sigma}$ and $\mathtt{traces}(\overline{\sigma''}) \subseteq \mathtt{traces}(\overline{\sigma})$, hence by Proposition 4.1 we have $\overline{\sigma''} \preceq \sigma'$ that is $\sigma'' \between_c \sigma'$. By the induction hypothesis we conclude that $P' \between_p Q$ hence $P \between_p Q$.

($Q \xrightarrow{\tau} Q'$) Assume $Q' \vdash \sigma''$, then from Lemma 4.4(a) we know that $\sigma' \preceq \sigma''$ and $\mathtt{traces}(\sigma'') \subseteq \mathtt{traces}(\sigma')$, hence by Proposition 4.1 we have $\overline{\sigma} \preceq \sigma''$ that is $\sigma \between_c \sigma''$. By the induction hypothesis we conclude that $P \between_p Q'$ hence $P \between_p Q$.

($P \xrightarrow{\alpha} P'$ **and** $Q \xrightarrow{\overline{\alpha}} Q'$) Assume that $P' \vdash \sigma''$ and $Q' \vdash \sigma'''$. From Lemma 4.4(b) know that $\overline{\sigma''} \preceq \overline{\sigma(\alpha)}$ and $\mathtt{traces}(\overline{\sigma''}) \subseteq \mathtt{traces}(\overline{\sigma(\alpha)})$, and by definition of dual contract we have $\overline{\sigma(\alpha)} = \overline{\sigma}(\overline{\alpha})$. Again from Lemma 4.4(b) we know that $\sigma'(\alpha) \preceq \sigma'''$ and $\mathtt{traces}(\sigma''') \subseteq \mathtt{traces}(\sigma'(\alpha))$. By Proposition 4.1 we have $\overline{\sigma''} \preceq \sigma'''$ that is $\sigma'' \between_c \sigma'''$. The computation starting from $P' \parallel Q'$ has length $n-1$, by the induction hypothesis we have $P' \between_p Q'$ so we conclude $P \between_p Q$. $\square$

Chapter 5

The Recursive Contract Language

In this chapter we extend the finite contract language with recursive contract definitions. According to that the subcontract and the compliance relations are also extended. As in the finite case, contracts are used to describe conversation protocol implemented by processes. In this case we consider CCS processes with recursion. Since CCS allows to define infinite states protocols whilst contracts are finite states, we provide two relations, called *underestimation* and *overestimation*, for verifying that a given contract overestimates the client's capabilities and that a contract underestimates the server's capabilities. Then we demonstrate that *if a contract overestimates the client process and a contract underestimates the server process and the two contracts comply, then the processes also comply.* In case of regular processes – that are processes without parallel composition and restriction – we show how to extrapolate contracts out of processes.

Structure of the chapter. Section 5.1 formally defines the recursive contract language along with *subcontract* and *compliance* relations. In Section 5.2 we show how to express conversation protocols defined by the Web Service Conversation Language WSCL in the contract language. Section 5.3 extends to recursive processes, the notion of process *compliance* defined in the previous chapter. Section 5.4 defines how processes can be estimated by contracts. It concludes demonstrating that complying approximations imply compliant processes. In Section 5.5 we show how to extrapolate a contract out of regular (finite state) processes.

5.1 Recursive contracts

The syntax of contracts uses an infinite set of *names* $\mathcal{N}$ ranged over by a, b, c, $\ldots$, a disjoint set of *co-names* $\overline{\mathcal{N}}$ ranged over by $\overline{a}$, $\overline{b}$, $\overline{c}$, $\ldots$, and an infinite set of recursion variables $\mathcal{X}$ ranged over by x, y, $\ldots$. We let $\overline{\overline{a}} = a$. Recursive contracts σ are defined by the grammar in Table 5.1.

Table 5.1: The recursive contract language.

σ ::=		**contracts**
	$\mathbf{0}$	(empty)
	$a.\sigma$	(input prefix)
	$\overline{a}.\sigma$	(output prefix)
	$\sigma + \sigma$	(external choice)
	$\sigma \oplus \sigma$	(internal choice)
	$\mathsf{rec}\ x.\,\sigma$	(recursive definition)
	x	(variable)

Recursive contracts extends finite contracts with contract variables x and recursive definitions $\mathsf{rec}\ x.\,\sigma$. This allow us to express infinite conversations having a regular structure. As usual we inductively define the set of free variables of a contract σ – $\mathtt{fv}(\sigma)$ – as: $\mathtt{fv}(\mathbf{0}) = \emptyset$, $\mathtt{fv}(a.\sigma) = \mathtt{fv}(\sigma)$, $\mathtt{fv}(\overline{a}.\sigma) = \mathtt{fv}(\sigma)$, $\mathtt{fv}(\sigma + \sigma') = \mathtt{fv}(\sigma) \cup \mathtt{fv}(\sigma')$, $\mathtt{fv}(\sigma \oplus \sigma') = \mathtt{fv}(\sigma) \cup \mathtt{fv}(\sigma')$, $\mathtt{fv}(\mathsf{rec}\ x.\,\sigma) = \mathtt{fv}(\sigma) \setminus \{x\}$, $\mathtt{fv}(x) = \{x\}$. We say that a contract σ is *closed* if and only if $\mathtt{fv}(\sigma) = \emptyset$. In what follows, if it is not explicitly specified, we always assume that contracts are closed.

The contracts $\mathbf{0}$, $a.\sigma$, $\overline{a}.\sigma$, $\sigma + \sigma$, and $\sigma \oplus \sigma'$ are as in the finite case. The contract $\mathsf{rec}\ x.\,\sigma$ defines a recursive contract. It allows to define arbitrarily long (and also infinite) conversation protocols between communicating parties. For instance the contract $\mathsf{rec}\ x.\,\mathtt{Login}.(\overline{\mathtt{Invalid}}.x \oplus (\mathtt{Continue} + \mathtt{End})$ describes the conversation protocol of a service accepting `Login`s. In case of a successful authentication the choice of either `Continue` or `End` is left to the client. In case of an unsuccesful authentication an error message $\overline{\mathtt{Invalid}}$ is sent and the server becomes available for another `Login`. It is worth to remark that protocols σ are always regular. For instance is not possible to define a protocol such

as $\texttt{Login}^n.\texttt{Logout}^n$ where the number of $\texttt{Login}$s and $\texttt{Logout}$s is the same for an unbound number n.

The syntax of recursive contracts also includes contracts such as $\mathsf{rec}\ x.\,x$. Such expressions, where the bound variable is not guarded by a prefix $a._$ or $\overline{a}._$ – unguarded recursion – are problematic because the correspondent equation has no finite solutions. We do not impose any restriction on the contract language because, when contracts are used to describe conversation protocol implemented by a CCS processes, unguarded contract may be necessary. For instance the (regular) CCS process $(\mathsf{rec}\ x.\,a.x \mid \mathsf{rec}\ x.\,\overline{a}.x) \setminus a$ implements the conversation protocol described by the contract $\mathsf{rec}\ x.\,x$. In the following we let $\mathbf{\Omega} \stackrel{\text{def}}{=} \mathsf{rec}\ x.\,x$ and, for each definition, we discuss how it behaves with respect to unguarded recursions.

As in the finite case, we use α to range over names and co-names and we omit the trailing $\mathbf{0}$.

5.1.1 Subcontract relation

We now extend the transition relation on contracts (Definition 4.1 of Chapter 4). Let $\sigma \downarrow \alpha$ be the least predicate on open contracts such that

$$\begin{array}{rl} \alpha.\sigma \downarrow \alpha & \\ \sigma \oplus \sigma' \downarrow \alpha & \text{if } \sigma \downarrow \alpha \text{ or } \sigma' \downarrow \alpha \\ \sigma + \sigma' \downarrow \alpha & \text{if } \sigma \downarrow \alpha \text{ or } \sigma' \downarrow \alpha \\ \mathsf{rec}\ x.\,\sigma \downarrow \alpha & \text{if } \sigma \downarrow \alpha \end{array}$$

We write $\sigma \not\downarrow \alpha$ when $\sigma \downarrow \alpha$ is false. For example $a.x \oplus b \downarrow a$ and $a.x \oplus b \not\downarrow c$. Also $x \not\downarrow a$ and $\mathbf{\Omega} \not\downarrow a$ for every a.

Definition 5.1 (Transition $\stackrel{\boldsymbol{\alpha}}{\longmapsto}$) *The* transition relation *of contracts, noted* $\stackrel{a}{\longmapsto}$, *is the*

least relation satisfying the rules:

$$\alpha.\sigma \overset{\alpha}{\longmapsto} \sigma$$

$$\frac{\sigma_1 \overset{\alpha}{\longmapsto} \sigma_1' \quad \sigma_2 \overset{\alpha}{\longmapsto} \sigma_2'}{\sigma_1 + \sigma_2 \overset{\alpha}{\longmapsto} \sigma_1' \oplus \sigma_2'} \qquad \frac{\sigma_1 \overset{\alpha}{\longmapsto} \sigma_1' \quad \sigma_2 \not\downarrow \alpha}{\sigma_1 + \sigma_2 \overset{\alpha}{\longmapsto} \sigma_1'}$$

$$\frac{\sigma_1 \overset{\alpha}{\longmapsto} \sigma_1' \quad \sigma_2 \overset{\alpha}{\longmapsto} \sigma_2'}{\sigma_1 \oplus \sigma_1' \overset{\alpha}{\longmapsto} \sigma_2 \oplus \sigma_2'} \qquad \frac{\sigma_1 \overset{\alpha}{\longmapsto} \sigma_1' \quad \sigma_2 \not\downarrow \alpha}{\sigma_1 \oplus \sigma_2 \overset{\alpha}{\longmapsto} \sigma_1'}$$

$$\frac{\sigma \overset{\alpha}{\longmapsto} \sigma'}{\mathsf{rec}\ x.\,\sigma \overset{\alpha}{\longmapsto} \sigma'\{^{\mathsf{rec}\ x.\,\sigma}/_{x}\}}$$

and closed under mirror cases for external and internal choices.

This transition relation extends the one of Chapter 4 to recursive terms. We remark that the transition relation of recursive contracts is defined by induction on the structure of the contract. In particular, the transition relation is also defined on open terms. For example $a.x \oplus b \overset{a}{\longmapsto} x$ and $x \oplus a.b \overset{a}{\longmapsto} b$. Other examples follow:

Example:

1. $\mathsf{rec}\ x.\,a.b.x + \mathbf{\Omega} \overset{a}{\longmapsto} b.\mathsf{rec}\ x.\,a.b.x$
2. $\mathsf{rec}\ x.\,a \oplus x \overset{a}{\longmapsto} \mathbf{0}$
3. $\mathsf{rec}\ x.\,a.b.x + \mathsf{rec}\ x.\,a.c.x \overset{a}{\longmapsto} b.\mathsf{rec}\ x.\,a.b.x \oplus c.\mathsf{rec}\ x.\,a.c.x$
4. $\mathsf{rec}\ x.\,(a.b + x) + \mathsf{rec}\ x.\,(a.c + x) \overset{a}{\longmapsto} b \oplus c$
5. $\mathsf{rec}\ x.\,(a.b.x + a.x) \overset{a}{\longmapsto} b.\mathsf{rec}\ x.\,(a.b.x + a.x) \oplus \mathsf{rec}\ x.\,(a.b.x + a.x)$

In the third example, the contract $\mathsf{rec}\ x.\,a.b.x + \mathsf{rec}\ x.\,a.c.x$ transits with a into $b.\mathsf{rec}\ x.\,a.b.x \oplus c.\mathsf{rec}\ x.\,a.c.x$. The situation is similar to finite contracts and describes the fact that when the activity a has been done, the communicating party is not yet aware of which part of the conversation has been selected. In CCS, the process $\mathsf{rec}\ x.\,a.b.x + \mathsf{rec}\ x.\,a.c.x$ has two transitions $\mathsf{rec}\ x.\,a.b.x + \mathsf{rec}\ x.\,a.c.x \xrightarrow{a} b.\mathsf{rec}\ x.\,a.b.x$ and $\mathsf{rec}\ x.\,a.b.x + \mathsf{rec}\ x.\,a.c.x \xrightarrow{a} c.\mathsf{rec}\ x.\,a.c.x$.

Proposition 5.1 $\stackrel{\alpha}{\longmapsto}$ *is deterministic (for every contract σ, there is always at most one contract σ' such that $\sigma \stackrel{\alpha}{\longmapsto} \sigma'$).*

We write $\sigma(\alpha)$ for the unique σ' such that $\sigma \stackrel{\alpha}{\longmapsto} \sigma'$. As in the finite case, we also define the set of traces of a contract σ, written $\mathtt{traces}(\sigma)$, has $\mathtt{traces}(\sigma) \stackrel{\text{def}}{=} \{\alpha_1 \cdots \alpha_n : \sigma \stackrel{\alpha_1}{\longmapsto} \cdots \stackrel{\alpha_n}{\longmapsto}\}$. It is easy to see that $\mathtt{traces}(\sigma) \subseteq \mathtt{traces}(\sigma')$ implies $\mathtt{traces}(\sigma(\alpha)) \subseteq \mathtt{traces}(\sigma'(\alpha))$.

Definition 5.2 (Contract Convergence) *Let $\sigma \Downarrow$, read σ* converges*, be the least predicate over contracts such that:*

$$
\begin{array}{ll}
\mathbf{0} \Downarrow & \\
\alpha.\sigma \Downarrow & \\
\sigma + \sigma' \Downarrow & \text{if } \sigma \Downarrow \text{ and } \sigma' \Downarrow \\
\sigma \oplus \sigma' \Downarrow & \text{if } \sigma \Downarrow \text{ and } \sigma' \Downarrow \\
\mathsf{rec}\ x.\,\sigma \Downarrow & \text{if } \sigma\{^{\mathsf{rec}\ x.\,\sigma}/_{x}\} \Downarrow
\end{array}
$$

We write $\sigma \not\Downarrow$, read σ diverges*, if not $\sigma \Downarrow$.*

The definition of contract convergence verifies whether a contract is unguarded or not. Unguarded contracts describe possibly divergent conversation protocols i.e. protocols that may loop forever without performing any action. For instance $a \oplus \mathbf{\Omega} \not\Downarrow$ and $\mathsf{rec}\ x.\, x + a \not\Downarrow$ because they describe conversations where either an input on a is offered or "nothing" is offered.

It stands that $\stackrel{\alpha}{\longmapsto}$ is insensitive to divergence. For instance $\mathbf{\Omega}$ and $\mathbf{0}$ have no transitions and $\mathsf{rec}\ x.\, a + x$ and a have the same transition $\stackrel{a}{\longmapsto}$ with $\mathbf{0}$ as successor. This is because $\stackrel{\alpha}{\longmapsto}$ describes the conversation from the perspective of the interacting party, which cannot observe the divergence. However there is an important difference between the contract a and the contract $\mathsf{rec}\ x.\, a + x$: a always perform the action a, whilst $\mathsf{rec}\ x.\, a + x$ may diverge without offering any action. That is, $a + \mathbf{\Omega}$ and $a \oplus \mathbf{\Omega}$ are equivalent to $a \oplus \mathbf{0}$ in the sense that everyone *may* accept a message on a. We take into account this aspect in the following definition of ready sets.

Definition 5.3 (Ready sets) *Let* $\sigma \searrow \mathrm{R}$*, read* σ has ready set R*, be the least relation such that:*

$$
\begin{array}{ll}
\mathbf{0} \searrow \emptyset & \\
\alpha.\sigma \searrow \{\alpha\} & \\
(\sigma + \sigma') \searrow \mathrm{R} \cup \mathrm{R}' & \text{if } \sigma \searrow \mathrm{R} \text{ and } \sigma' \searrow \mathrm{R}' \\
(\sigma \oplus \sigma') \searrow \mathrm{R} & \text{if either } \sigma \searrow \mathrm{R} \text{ or } \sigma' \searrow \mathrm{R} \\
\mathsf{rec}\ x.\,\sigma \searrow \mathrm{R} & \text{if } \sigma\{{}^{\mathsf{rec}\ x.\,\sigma}/_{x}\} \searrow \mathrm{R} \\
\sigma \searrow \emptyset & \text{if } \sigma \not\Downarrow
\end{array}
$$

The definition of ready sets in the recursive case is the extension of the one of the finite contract language (Definition 4.2 of Chapter 4) with two clauses. The first clause says a contract has a ready set R only if its unfolding has the ready set R. The second clause says that diverging contracts always have also an empty ready set because they may diverge without offering any action to the parties. For instance, $(\mathsf{rec}\ x.\, a.x) + b$ has $\{a, b\}$ as unique ready set; $\mathsf{rec}\ x.\, a.x \oplus \mathsf{rec}\ y.\, b.y$ has ready sets $\{a\}$ and $\{b\}$. As regards unguarded recursion, $\mathbf{\Omega} \searrow \emptyset$ and $\mathbf{\Omega} \oplus a$ has two ready sets: $\{a\}$ and $\emptyset$. Similarly, $\mathsf{rec}\ x.\, \mathbf{\Omega} + a.x \searrow \emptyset$ and $\mathsf{rec}\ x.\, \mathbf{\Omega} + a.x \searrow \{a\}$.

Definition 5.4 (Subcontracts) *A* subcontract relation $\mathcal{S}$ *is a relation on contracts such that* $\sigma_1 \mathcal{S} \sigma_2$ *implies:*

1. *if* $\sigma_2 \searrow \mathrm{R}_2$ *then* $\sigma_1 \searrow \mathrm{R}_1$ *with* $\mathrm{R}_1 \subseteq \mathrm{R}_2$*,*
2. *if* $\sigma_1 \overset{\alpha}{\longmapsto} \sigma_1'$ *and* $\sigma_2 \overset{\alpha}{\longmapsto} \sigma_2'$ *then* $\sigma_1' \mathcal{S} \sigma_2'$*.*

Let $\preceq$ *be the largest subcontract relation. Let* $\sigma_1 \simeq \sigma_2$*, called* contract equivalence*, if both* $\sigma_1 \preceq \sigma_2$ *and* $\sigma_2 \preceq \sigma_1$*.*

The definition of subcontract is the same as for the finite case. We recall it for readability sake. The above definition takes into account divergence in the first item. For instance one may verify that $a + \mathbf{\Omega} \preceq \mathsf{rec}\ x.\, a + x$ and $a \not\preceq \mathsf{rec}\ x.\, a + x$ because of the $\mathsf{rec}\ x.\, a + x \searrow \emptyset$. One may also verify the following relations:

1. $\mathsf{rec}\ x.\, a.x \simeq \mathsf{rec}\ x.\, a.a.x$
2. $\mathbf{\Omega} \simeq \mathbf{0}$
3. $\mathbf{\Omega} \preceq \sigma$

4. $a \oplus \mathbf{0} \simeq a + \mathbf{\Omega} \simeq \mathsf{rec}\ x.\, a + x \simeq a \oplus \mathbf{\Omega}$

5. $\mathsf{rec}\ x.\, a.x \oplus b.x \preceq \mathsf{rec}\ x.\, a.b.x$

6. $\mathsf{rec}\ x.\, a.b.x \preceq \mathsf{rec}\ x.\, a.x + b.x$

As in the finite case, $\preceq$ is not transitive. For instance we notice that $\mathsf{rec}\ x.\, a.x \oplus b.c.x \preceq \mathsf{rec}\ x.\, a.x$ and $\mathsf{rec}\ x.\, a.x \preceq \mathsf{rec}\ x.\, a.x + b.x$ do not entail $\mathsf{rec}\ x.\, a.x \oplus b.c.x \preceq \mathsf{rec}\ x.\, a.x + b.x$. However transitivity holds if traces are included. The following Lemma and Propositions correspond to Lemma 4.1, Proposition 4.2, and Proposition 4.3 of Chapter 4. The proof are identical thus omitted.

Lemma 5.1 *If $\sigma_1 \preceq \sigma_2$ and $\sigma_2 \preceq \sigma_3$ and either (a) $\mathtt{traces}(\sigma_1) \subseteq \mathtt{traces}(\sigma_2)$ or (b) $\mathtt{traces}(\sigma_3) \subseteq \mathtt{traces}(\sigma_2)$ then $\sigma_1 \preceq \sigma_3$.*

Proposition 5.2 *$\sigma \oplus \sigma' \preceq \sigma$.*

Proposition 5.3 *If $\sigma \preceq \sigma' \oplus \sigma''$ then $\sigma \preceq \sigma'$ and $\sigma \preceq \sigma''$.*

The notion of *dual contract* in the recursive case is an extension of the one of the previous chapter. Therefore in order to avoid fallible transformations we first need to rewrite the contract in normal form and then we extrapolate the dual contract by inverting inputs and outputs, internal choices and external choice. In what follows we let A be an injective mapping from contracts to names with the following operation:

$$A + [\sigma \mapsto x] \quad = \quad \begin{cases} A & \text{if } \sigma \in \mathtt{dom}(A) \\ A \cup \{(\sigma, x)\} & \text{if } \sigma \notin \mathtt{dom}(A) \text{ and } x \notin \mathtt{cod}(A) \end{cases}$$

We also assume that the name x, related to the contract σ, is unique and uniquely determined by σ.

Definition 5.5 (Normed Contract) *Let*

$$\mathtt{nc}_A(\sigma) = \begin{cases} \mathsf{rec}\ x.\ \bigoplus_{\sigma \searrow \mathrm{R}} \sum_{\alpha \in \mathrm{R}} \alpha.\mathtt{nc}_{A'}(\sigma(\alpha)) & \text{if } A' = A + [\sigma \mapsto x],\ x \in \mathtt{fv}(\mathtt{nc}_{A'}(\sigma(\alpha))) \\ \bigoplus_{\sigma \searrow \mathrm{R}} \sum_{\alpha \in \mathrm{R}} \alpha.\mathtt{nc}_{A'}(\sigma(\alpha)) & \text{if } A' = A + [\sigma \mapsto x],\ x \notin \mathtt{fv}(\mathtt{nc}_{A'}(\sigma(\alpha))) \\ A(\sigma) & \text{if } \sigma \in \mathtt{dom}(A) \end{cases}$$

where, by convention, we have $\sum_{\alpha \in \emptyset} = \mathbf{0}$. The normed contract of σ, noted $\mathtt{nc}(\sigma)$, is defined as $\mathtt{nc}_{\emptyset}(\sigma)$

The *normed contract* of σ, is defined as in the finite case using ready sets and the transition relation $\stackrel{\alpha}{\longmapsto}$. The two cases of the definition account for the need of the recursion for the normed contract of the continuation. Then, a recursive contract $\mathsf{rec}\ x. _$ is introduced when necessary and the map $\sigma \mapsto x$ is added to A. After that, the normed contract of the continuation $\sigma(\alpha)$ is computed. If the normed contract of σ has been already met – $A(\sigma)$ is defined – we close returning its name $A(\sigma)$. We note that the existence of the normal form of a contract is due to the fact that the transition system of $\stackrel{\alpha}{\longmapsto}$ has finitely many different states. Said otherwise the inductive parameter is the number of distinct states of the transition relation $\longmapsto$ We also note that – in case of finite contracts – this definition of normed contract coincides with the one in Chapter 4.

Example:

1. $\mathtt{nc}(\mathbf{\Omega}) = \mathbf{0}$

2. $\mathtt{nc}(\mathsf{rec}\ x. a + x) = a \oplus \mathbf{0}$

3. $\mathtt{nc}(a.b + \mathsf{rec}\ x. a.c.x) = a.(b \oplus c.\mathsf{rec}\ x. a.c.x)$, indeed

$$\begin{aligned}
\mathtt{nc}_{\emptyset}(a.b + \mathsf{rec}\ x. a.c.x) &= a.(\mathtt{nc}_{[\sigma_1 \mapsto y]}(b \oplus c.\mathsf{rec}\ x. a.c.x)) \\
&= a.(b \oplus c.\mathtt{nc}_{[\sigma_1 \mapsto y, \sigma_2 \mapsto z]}(\mathsf{rec}\ x. a.c.x)) \\
&= a.(b \oplus c.\mathsf{rec}\ x. a.\mathtt{nc}_{\begin{bmatrix}\sigma_1 \mapsto y, \sigma_2 \mapsto z, \\ \sigma_3 \mapsto x\end{bmatrix}}(c.\mathsf{rec}\ x. a.c.x)) \\
&= a.(b \oplus c.\mathsf{rec}\ x. a.c.\mathtt{nc}_{\begin{bmatrix}\sigma_1 \mapsto y, \sigma_2 \mapsto z, \\ \sigma_3 \mapsto x, \sigma_4 \mapsto w\end{bmatrix}}(\mathsf{rec}\ x. a.c.x)) \\
&= a.(b \oplus c.\mathsf{rec}\ x. a.c.x)
\end{aligned}$$

with $\sigma_1 = a.b + \mathsf{rec}\ x. a.c.x$, $\sigma_2 = b \oplus c.\mathsf{rec}\ x. a.c.x$, $\sigma_3 = \mathsf{rec}\ x. a.c.x$, $\sigma_4 = c.\mathsf{rec}\ x. a.c.x$

4. $\mathtt{nc}(\mathsf{rec}\ x. a.b.x + \mathsf{rec}\ y. a.c.y) = a.(b.\mathsf{rec}\ x. a.b.x \oplus c.\mathsf{rec}\ x. a.c.x)$

To show, as in the finite case, the correspondance between σ and its normed contract we need the following preliminar.

Proposition 5.4 *Let $\sigma' \notin \mathtt{dom}(A)$. $\mathtt{nc}_{A+[\sigma \mapsto x]}(\sigma')\{^{\mathtt{nc}_A(\sigma)}/_{x}\} = \mathtt{nc}_A(\sigma')$.*

Proof: We proceed by induction on the structure of $\mathtt{nc}_{A+[\sigma \mapsto x]}(\sigma')$. The case $\mathtt{nc}_{A+[\sigma \mapsto x]}(\sigma') = \mathbf{0}$ is immediate. In case of $\mathtt{nc}_{A+[\sigma \mapsto x]}(\sigma') = x$, by definition of $\mathtt{nc}_A(\cdot)$, $\sigma' = \sigma$ and we conclude. In case of $\mathtt{nc}_{A+[\sigma \mapsto x]}(\sigma') = y \neq x$, by definition of $\mathtt{nc}_A(\cdot)$, $A(\sigma') = y$ and $\sigma' \neq \sigma$.

Then $\mathtt{nc}_{A+[\sigma\mapsto x]}(\sigma')\{^{\mathtt{nc}_A(\sigma)}/_x\} = y$ and $\mathtt{nc}_A(\sigma') = y$. Hence we conclude. In the inductive case, by definition of $\mathtt{nc}_{A+[\sigma\mapsto x]}(\cdot)$, we distinguish two subcases depending on which rule of $\mathtt{nc}_{A+[\sigma\mapsto x]}(\sigma')$ has been applied.

– If $\mathtt{nc}_{A+[\sigma\mapsto x]}(\sigma') = \bigoplus_{\sigma'\searrow\mathrm{R}} \sum_{\alpha\in\mathrm{R}} \alpha.\sigma''$ with $\sigma'' = \mathtt{nc}_{A+[\sigma\mapsto x]+[\sigma'\mapsto y]}(\sigma'(\alpha))$ and $y \notin \mathtt{fv}(\sigma'')$ then, by the inductive hypothesis, $\sigma''\{^{\mathtt{nc}_A(\sigma)}/_x\} = \mathtt{nc}_{A+[\sigma'\mapsto y]}(\sigma')$. By $\sigma' \notin \mathtt{dom}(A)$ we have $y \notin \mathtt{fv}(\sigma''\{^{\mathtt{nc}_A(\sigma)}/_x\})$ thus $y \notin \mathtt{fv}(\mathtt{nc}_{A+[\sigma'\mapsto y]}(\sigma'))$ and $\mathtt{nc}_A(\sigma') = \bigoplus_{\sigma'\searrow} \sum_{\alpha\in\mathrm{R}} \alpha.\mathtt{nc}_{A+[\sigma'\mapsto y]}(\sigma'(\alpha))$. We conclude by the inductive hypothesis.

– If $\mathtt{nc}_{A+[\sigma\mapsto x]}(\sigma') = \mathsf{rec}\ y.\ \bigoplus_{\sigma'\searrow\mathrm{R}} \sum_{\alpha\in\mathrm{R}} \alpha.\sigma''$ with $\sigma'' = \mathtt{nc}_{A+[\sigma\mapsto x]+[\sigma'\mapsto y]}(\sigma'(\alpha))$ and $y \in \mathtt{fv}(\sigma'')$ then, by the inductive hypothesis, $\sigma''\{^{\mathtt{nc}_A(\sigma)}/_x\} = \mathtt{nc}_{A+[\sigma'\mapsto y]}(\sigma')$. By $y \in \mathtt{fv}(\sigma'')$, $y \in \mathtt{fv}(\sigma''\{^{\mathtt{nc}_A(\sigma)}/_x\})$ and $y \in \mathtt{fv}(\mathtt{nc}_{A+[\sigma'\mapsto y]}(\sigma'))$ and $\mathtt{nc}_A(\sigma') = \mathsf{rec}\ y.\ \bigoplus_{\sigma'\searrow} \sum_{\alpha\in\mathrm{R}} \alpha.\mathtt{nc}_{A+[\sigma'\mapsto y]}(\sigma'(\alpha))$. We conclude by the inductive hypothesis. □

Proposition 5.5

1. $\mathtt{nc}(\sigma) \Downarrow$*;*

2. $\sigma \searrow \mathrm{R}$ *if and only if* $\mathtt{nc}(\sigma) \searrow \mathrm{R}$*;*

3. $\sigma \overset{\alpha}{\longmapsto} \sigma(\alpha)$ *if and only if* $\mathtt{nc}(\sigma) \overset{\alpha}{\longmapsto} \mathtt{nc}(\sigma(\alpha))$*.*

Proof: Item 1 follows because $\mathtt{nc}$ always introduces variables underneath a prefix $\alpha._$.

Item 2 states that σ and its normed contract have the same ready sets and it is also an immediate consequence of the definition.

Item 3 ($\Leftarrow$) is immediate. As regards ($\Rightarrow$), by the hypothesis $\sigma \overset{\alpha}{\longmapsto} \sigma(\alpha)$ we have $\sigma \searrow \mathrm{R}\cup\{\alpha\}$. Then, by definition of $\mathtt{nc}_A(\cdot)$, we have two cases. The first case implies $\mathtt{nc}(\sigma) \overset{\alpha}{\longmapsto} \mathtt{nc}_{[\sigma\mapsto x]}(\sigma(\alpha))$ with $x \notin \mathtt{fv}(\mathtt{nc}_{[\sigma\mapsto x]}(\sigma(\alpha)))$. Hence we conclude $\mathtt{nc}_{[\sigma\mapsto x]}(\sigma(\alpha)) = \mathtt{nc}(\sigma(\alpha))$. The second case is $\mathtt{nc}(\sigma) \overset{\alpha}{\longmapsto} \mathtt{nc}_{[\sigma\mapsto x]}(\sigma(\alpha))\{^{\mathtt{nc}(\sigma)}/_x\}$ with $x \in \mathtt{fv}(\mathtt{nc}_{[\sigma\mapsto x]}(\sigma(\alpha)))$. Hence, by Proposition 5.4, $\mathtt{nc}_{[\sigma\mapsto x]}(\sigma(\alpha))\{^{\mathtt{nc}(\sigma)}/_x\} = \mathtt{nc}(\sigma(\alpha))$ and we conclude. □

As in the finite case the equivalence between a contract and its normal form follows by Proposition 5.5.

Lemma 5.2 *$\sigma \simeq \texttt{nc}(\sigma)$ and $\texttt{traces}(\sigma) = \texttt{traces}(\texttt{nc}(\sigma))$*

Definition 5.6 (Dual contract) *Let*

$$\overline{\sigma}_A = \begin{cases} \mathsf{rec}\ x.\ \sum_{\sigma \searrow \mathrm{R}} \bigoplus_{\alpha \in \mathrm{R}} \overline{\alpha}.\overline{\sigma(\alpha)}_{A'} & \text{if } A' = A + [\sigma \mapsto x],\ x \in \texttt{fv}(\overline{\sigma(\alpha)}_{A'}) \\ \sum_{\sigma \searrow \mathrm{R}} \bigoplus_{\alpha \in \mathrm{R}} \overline{\alpha}.\overline{\sigma(\alpha)}_{A'} & \text{if } A' = A + [\sigma \mapsto x],\ x \notin \texttt{fv}(\overline{\sigma(\alpha)}_{A'}) \\ A(\sigma) & \text{if } \sigma \in \texttt{dom}(A) \end{cases}$$

where, by convention, we have $\bigoplus_{\alpha \in \emptyset} = \mathbf{0}$. The dual contract *of σ, noted $\overline{\sigma}$, is defined as $\overline{\sigma}_\emptyset$*

The dual contract is defined similarly to the normal form but inverting internal choices with external choices and inputs with outputs. Then, two cases of the definition depend on the need of the recursive definition for the dual of the continuation. Also for the dual contract, it is easy to see that – in case of finite contracts – this definition coincides with the one in Chapter 4.
Example:

1. $\overline{\mathbf{\Omega}} = \mathbf{0}$
2. $\overline{a + \mathbf{\Omega}} = \overline{a \oplus \mathbf{\Omega}} = \overline{a \oplus \mathbf{0}} = \overline{\mathsf{rec}\ x.\, a + x} = \overline{a} + \mathbf{0}$
3. $\overline{(a + b) \oplus \mathbf{\Omega}} = \overline{a + b + \mathbf{\Omega}} = \overline{a} \oplus \overline{b} + \mathbf{0}$
4. Let $\sigma = \mathsf{rec}\ x.\, a.b.x + a.c.x$.

$$\begin{aligned} \overline{\sigma} &= \mathsf{rec}\ x.\, \overline{a}.(\overline{b.\sigma \oplus c.\sigma}_{[\sigma \mapsto x]}) \\ &= \mathsf{rec}\ x.\, \overline{a}.(\overline{b}.\overline{\sigma}_{[\sigma \mapsto x]} + \overline{c}.\overline{\sigma}_{[\sigma \mapsto x]}) \\ &= \mathsf{rec}\ x.\, \overline{a}.(\overline{b}.x + \overline{c}.x) \end{aligned}$$

$$\begin{aligned} \overline{a + \sigma} &= a.(\overline{\mathbf{0} \oplus b.\sigma \oplus c.\sigma}) \\ &= a.(\mathbf{0} + \overline{b.\sigma} + \overline{c.\sigma}) \\ &= a.(\mathbf{0} + \overline{b}.\overline{\sigma} + \overline{c}.\overline{\sigma} \\ &= a.(\mathbf{0} + \overline{b}.\mathsf{rec}\ x.\, \overline{a}.(\overline{b}.x + \overline{c}.x) + \overline{c}.\mathsf{rec}\ x.\, \overline{a}.(\overline{b}.x + \overline{c}.x) \end{aligned}$$

Examples 1,2,3 show the duals of diverging contracts. Since $\sigma \not\Downarrow$ implies $\sigma \searrow \emptyset$, the dual of a diverging contract is either $\mathbf{0}$ (example 1) or an external choice with $\mathbf{0}$ (examples

2,3). In the example 3, we remind that $(\mathsf{rec}\ x.\, a + x)(a) = \mathbf{0}$. It follows that the dual of a divergent contract is convergent. This is not surprising, since the dual of a contract is used to obtain an abstract description of the server's protocol, a divergent client is assimilable to an inactive client $\mathbf{0}$ because both do not require any particular behavior to the server. Example 4 shows the dual of two recursive contracts where the map A is used to ensure termination. In particular, the termination is due to the finiteness of the set of states generated from a contract by $\longmapsto$.

Similarly to the finite case (see Proposition 4.5), the following propositions hold:

Proposition 5.6

1. $\overline{\sigma} \Downarrow$;
2. $\overline{\sigma} \searrow \mathrm{R}$ *if and only if* $\{\mathrm{R}_1, \ldots, \mathrm{R}_n\} = \{\mathrm{R}' : \sigma \searrow \mathrm{R}'\}$, $\mathrm{R} = \bigcup_{i \in 1..n} \overline{\alpha_i}, \alpha_i \in \mathrm{R}_i$;
3. *if* $\overline{\sigma} \searrow \emptyset$ *then* $\sigma \searrow \mathrm{R}$ *implies* $\mathrm{R} = \emptyset$;
4. *if* $\sigma \overset{\alpha}{\longmapsto} \sigma(\alpha)$ *then* $\overline{\sigma} \overset{\overline{\alpha}}{\longmapsto} \overline{\sigma(\alpha)}$.

Proof: Items 1, 2, 3 are immediate by definition of $\overline{\sigma}$. The proof of item 4 is a straightforward adaptation of the proof of Proposition 5.5.3. □

Lemma 5.3 $\overline{\sigma} \preceq \overline{\sigma \oplus \sigma'}$.

Proof: See Chapter 4 Lemma 4.3. □

5.1.2 Contract compliance

Every preliminary notion has been set for the definition of contract compliance for recursive contracts.

Definition 5.7 (Contract compliance) $\sigma \between_c \sigma'$ *if and only if* $\overline{\sigma} \preceq \sigma'$.

The notion of compliance is the same of the finite case. A client with contract σ will interact successfully with every service with contract σ' provided $\sigma \between_c \sigma'$. In the finite case, the communication is successful only if the client termination is guaranteed. In the infinite case the termination constraint is relaxed and we require clients to progress. That is, querying a repository for services compliant with $\mathsf{rec}\ x.\, a.x \oplus b.x$, may return services

implementing the conversation protocols $\mathsf{rec}\ x.\,\overline{a}.x + \overline{b}.x$, $\mathsf{rec}\ x.\,\overline{a}.x + \overline{b}.x + c.x$ etc.... On the contrary $\mathsf{rec}\ x.\,\overline{a}.x \oplus \overline{b}.x$ cannot be returned. Indeed,

$$\begin{array}{rrcl} & \overline{\mathsf{rec}\ x.\,a.x \oplus b.x} & = & \mathsf{rec}\ x.\,\overline{a}.x + \overline{b}.x \\ \text{and} & \mathsf{rec}\ x.\,\overline{a}.x + \overline{b}.x & \preceq & \mathsf{rec}\ x.\,\overline{a}.x + \overline{b}.x + c.x \\ \text{but} & \mathsf{rec}\ x.\,\overline{a}.x + \overline{b}.x & \not\preceq & \mathsf{rec}\ x.\,a.x \oplus b.x \end{array}$$

When a client with contract $\mathbf{\Omega}$ queries the repository, every service can be returned. Indeed $\overline{\mathbf{\Omega}} = \mathbf{0}$ and $\mathbf{0} \preceq \sigma$ for every σ. This is because the client protocol does not need to interact at all with the service thus no particular capability is required at all.

Another case is when a client with contract $\mathbf{\Omega} \oplus \sigma$ queries the repository. We note that any client with contract $\mathbf{\Omega} \oplus \sigma$ either diverges – that from the perspective of the interacting party is equivalent to be inactive – or behaves as σ. According to our definition of dual, we have $\overline{\mathbf{\Omega} \oplus \sigma} = \overline{\mathbf{0} \oplus \sigma} = \overline{\sigma}$ thus the repository returns services compliant with σ.

A slightly different case is when the client has contract $\mathbf{\Omega} + \sigma$. Indeed, $\mathbf{\Omega} + \sigma$ describes a client that behaves either as $\mathbf{\Omega}$ or as σ according to the server's choice. In our definition of dual contract also $\overline{\mathbf{\Omega} + \sigma} = \overline{\sigma}$ thus requiring service to be compliant with σ. For instance if $\sigma = a$ then $\overline{a}$, $\overline{a} + b$ are compatible, whilst services such as $\mathbf{0}$ and b are not compliant with $\mathbf{\Omega} + a$. In Section 5.3 we will see that the process implementing $\mathbf{\Omega} + a$ (i.e. $\mathsf{rec}\ x.\,\tau.x + a$) is compliant with $\mathbf{0}$. Thus, as in the finite case, the notion of contract compliance is more demanding than process compliance.

Proposition 5.7 *If $\sigma \between_c \sigma'$ and $\sigma \overset{\alpha}{\longmapsto}$ and $\sigma' \overset{\overline{\alpha}}{\longmapsto}$ then $\sigma(\alpha) \between_c \sigma'(\overline{\alpha})$.*

Proof: By the hypotheses we have $\overline{\sigma} \preceq \sigma'$ and $\overline{\sigma} \overset{\overline{\alpha}}{\longmapsto} \overline{\sigma(\alpha)}$. Since $\sigma' \overset{\overline{\alpha}}{\longmapsto} \sigma'(\overline{\alpha})$ we must have $\overline{\sigma(\alpha)} \preceq \sigma'(\overline{\alpha})$. Hence we conclude. $\square$

Proposition 5.8 *The following proposition hold:*

1. *if $\sigma \oplus \sigma' \between_c \sigma''$ then $\sigma \between_c \sigma''$ and $\sigma' \between_c \sigma''$;*

2. *if $\sigma \between_c \sigma' \oplus \sigma''$ then $\sigma \between_c \sigma'$ and $\sigma \between_c \sigma''$.*

Proof: In case 1 we have to prove $\overline{\sigma} \preceq \sigma''$ and $\overline{\sigma'} \preceq \sigma''$. By Lemma 5.3 $\overline{\sigma} \preceq \overline{\sigma \oplus \sigma'}$ and $\overline{\sigma'} \preceq \overline{\sigma \oplus \sigma'}$. Since $\mathtt{traces}(\overline{\sigma}) \subseteq \mathtt{traces}(\overline{\sigma \oplus \sigma'})$, we conclude by Proposition 5.4. Case 2 is an immediate consequence of Proposition 5.3. $\square$

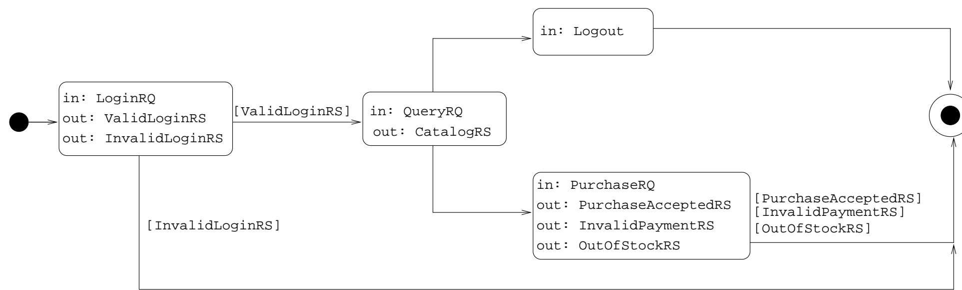


Figure 5.1: Contract of a simple e-commerce service as a WSCL diagram.

5.2 Conversations in WSCL

The WSDL message exchange patterns cover only the simplest forms of interaction between a client and a service. More involved forms of interactions, in particular stateful interactions, cannot be captured if not as informal annotation within the WSDL interface. The Web service conversation language WSCL [BBB+02, BKL01] provides a more general specification language for describing complex *conversations* between two communicating parties, by means of an activity diagram. The diagram is basically made of *interactions* which are connected with each other by means of *transitions*. An interaction is a basic one-way or two-way communication between the client and the server. Two-way communications are just a shorthand for two sequential one-way interactions. Each interaction has a *name* and a list of *document types* that can be exchanged during its execution. A transition connects a *source* interaction with a *destination* interaction. A transition may be *labeled* by a document type if it is active only when a message of that specific document type was exchanged during the previous interaction.

A finite WSCL diagram

Below we encode the contract σ of a simplified e-commerce service (Figure 5.1) where the client is required to login before it can issue a query and thus receive a catalog. From this point on, the client can decide whether to purchase an item from the catalog or to logout and leave. In case of purchase, the service may either report that the purchase was

successful, or that the item is out-of-stock, or that the client's payment was refused:

$$\sigma = \texttt{LoginRQ}.(\overline{\texttt{InvalidLoginRS}}.\overline{\texttt{End}} \oplus \overline{\texttt{ValidLoginRS}}.\texttt{QueryRQ}.\overline{\texttt{CatalogRS}}.(\\ \texttt{Logout}.\overline{\texttt{End}} + \texttt{PurchaseRQ}.(\\ \overline{\texttt{PurchaseAcceptedRS}}.\overline{\texttt{End}} \oplus \overline{\texttt{InvalidPaymentRS}}.\overline{\texttt{End}} \oplus \overline{\texttt{OutOfStockRS}}.\overline{\texttt{End}})))$$

Notice that unlabeled transitions in Figure 5.1 correspond to external choices in σ, whereas labeled transitions correspond to internal choices. It is also interesting to notice that WSCL explicitly accounts for a termination message (called "empty" in the WSCL specification, the final interaction on the right end in Figure 5.1) that is used for modeling the end of a conversation. The presence of this termination message finds a natural justification in our formal contract language, as explained above.

Now assume that the service is extended with a booking capability, so that after looking at the catalog the client may book an item to be bought at some later time. The contract of the service would change to σ' as follows:

$$\sigma' \stackrel{\text{def}}{=} \ldots \texttt{Logout}.\overline{\texttt{End}} + \texttt{Book}.\overline{\texttt{End}} + \texttt{Purchase}.(\ldots)$$

We notice that $\sigma \preceq \sigma'$ and $\texttt{traces}(\sigma) \subseteq \texttt{traces}(\sigma')$, that is σ' offers more capabilities than σ.

An infinite WSCL diagram

The finite contract language allows us to encode WSDL message exchange pattern that define finite conversation protocols. However the finite contract language is not powerful enough to encode WSCL's activity diagrams containing cycles. Figure 5.2 describes the complete example presented in [BKL01]. It is clear that this diagram, and actually any WSCL activity diagram, having a finite number of states defines a regular behavior thus suitable to be encoded in our language. We only need to create recursive definitions for those states that are part of a cycle. (Let us use italic font for contracts, typewriter for

actions.)

$$
\begin{aligned}
\sigma &= Registration + Login \\
Registration &= \texttt{RegistrationRQ}.\overline{\texttt{RegistrationRS}}.Login \\
Login &= \mathsf{rec}\ x.\,\texttt{LoginRQ}.(\overline{\texttt{InvalidLoginRS}}.(Registration + x)\oplus \\
&\qquad\qquad\qquad\qquad \overline{\texttt{ValidLoginRS}}.(CatalogInquiry + Purchase)) \\
CatalogInquiry &= \mathsf{rec}\ y.\,\texttt{CatalogRQ}.(\texttt{CatalogRS}.y + Quote + Logout) \\
Quote &= \texttt{QuoteRQ}.\texttt{QuoteRS}.(\texttt{CatalogInquiry} + Purchase + Logout) \\
Purchase &= \mathsf{rec}\ z.\,\texttt{PurchaseRQ}.(\texttt{PurchaseAcceptedRS}.Shipping\oplus \\
&\qquad\qquad\qquad\qquad \texttt{InvalidPaymentRS}.(z + \overline{\texttt{End}})\oplus \\
&\qquad\qquad\qquad\qquad \texttt{OutOfStockRS}.\overline{\texttt{End}}) \\
Shipping &= \overline{\texttt{OutShippingInformation}}.\overline{\texttt{End}} \\
Logout &= \texttt{LogoutMessage}.\overline{\texttt{End}}
\end{aligned}
$$

5.3 Process Compliance

Process compliance relates clients and service processes. A client is compliant with a service if the client cannot deadlock interacting with the service. Therefore, as in the finite case compliance induce a satisfiability property for the client. In order to formalize compliance we extend finite CCS processes with recursive definitions and we define their dynamics. Then we show how to verify that a contract properly describes the protocol implemented by a certain process. In particular two relations are presented. The first one, called *understimation*, is used to verify that a contract is an under-specification of the process behavior. The second one, called *overestimation*, verifies that a contract is an over-specification of the process behavior. Indeed either an overestimation of the server's capabilities or an underestimation of the client capabilities may clearly jeopardize the client satisfaction.

Approximations are needed because CCS with recursion, parallel composition and hiding is Turing complete [BGZ04] – a CCS process can simulate the Random Access Machines – whilst contracts cannot. Indeed, since contracts can only describes regular behaviors, the contract of a non-regular process must estimate the protocol implemented by the process. Such an estimation must be safe in the sense that it must guarantee the termination

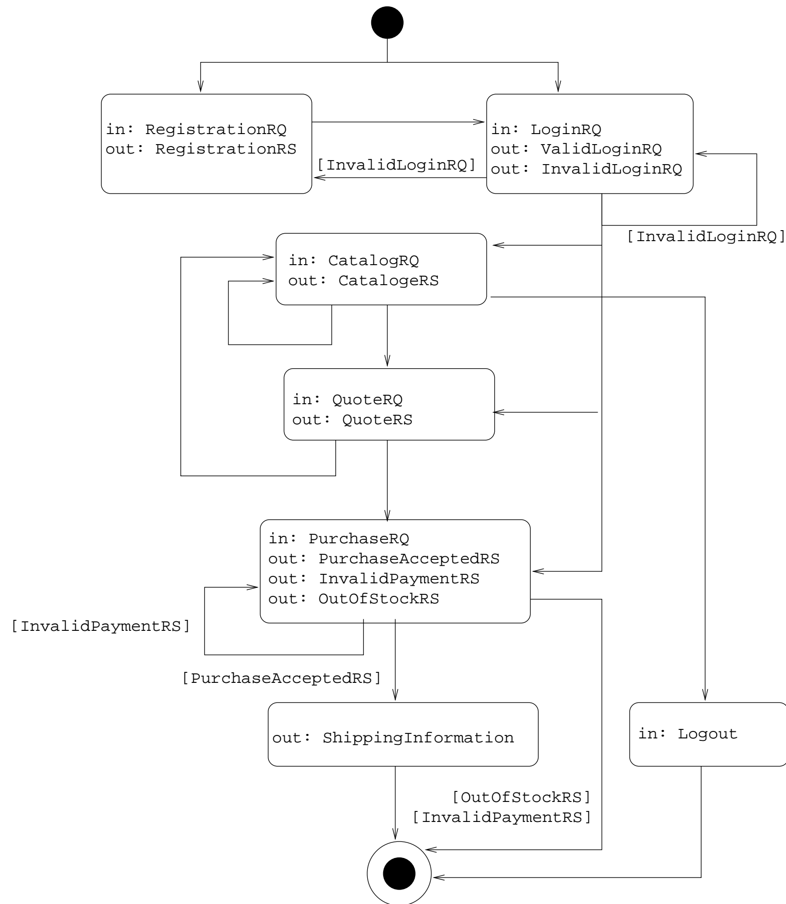


Figure 5.2: Contract of a e-commerce service as a WSCL diagram with cycles.

of clients. For regular processes, we can derive contracts without approximations. As in the finite case, we extrapolate contracts out of (regular) processes and we show that the resulting contract is both an overestimation and an underestimation of the conversation protocol implemented by the server. For this reason such a derivation is suitable to be used for querying and publishing services.

Definition 5.8 *Recursive* CCS *processes P are defined by the following grammar:*

P ::=		**processes**	μ ::=		**actions**
	$\mathbf{0}$	*(inaction)*		a	*(input)*
	$\mu.P$	*(action)*		$\overline{a}$	*(output)*
	$P \setminus a$	*(hiding)*		τ	*(silent)*
	$P \mid P$	*(parallel)*			
	$P + P$	*(choice)*			
	$\mathsf{rec}\ x.\, P$	*(definition)*			
	x	*(variable)*			

The transition relation *of processes, noted $\xrightarrow{\mu}$, is the least relation satisfying the rules:*

$$\text{(ACT)}\quad \mu.P \xrightarrow{\mu} P \qquad \text{(RES)}\quad \frac{P \xrightarrow{\mu} Q \quad \mu \notin \{a, \overline{a}\}}{P \setminus a \xrightarrow{\mu} Q \setminus a} \qquad \text{(PAR)}\quad \frac{P \xrightarrow{\mu} Q}{P \mid R \xrightarrow{\mu} Q \mid R}$$

$$\text{(COM)}\quad \frac{P \xrightarrow{\alpha} P' \quad Q \xrightarrow{\overline{\alpha}} Q'}{P \mid Q \xrightarrow{\tau} P' \mid Q'} \qquad \text{(REC)}\quad \frac{P\{^{\mathsf{rec}\ x.\, P}/_{x}\} \xrightarrow{\mu} P'}{\mathsf{rec}\ x.\, P \xrightarrow{\mu} P'} \qquad \text{(CHOICE)}\quad \frac{P \xrightarrow{\mu} P'}{P + Q \xrightarrow{\mu} P'}$$

The transitions of $P \mid Q$ and $P + Q$ have mirror cases that have been omitted.

We write $\overset{\tau}{\Longrightarrow}$ for $\xrightarrow{\tau}^{*}$, $\overset{\alpha}{\Longrightarrow}$ for $\xrightarrow{\tau}^{*}\xrightarrow{\alpha}\xrightarrow{\tau}^{*}$ and $P \overset{\mu}{\not\longrightarrow}$ if there not exist P' such that $P \xrightarrow{\mu} P'$.

As usual we inductively define the set of free variables of a process $\mathtt{fv}(P)$ as: $\mathtt{fv}(\mathbf{0}) = \emptyset$, $\mathtt{fv}(\alpha.P) = \mathtt{fv}(P \setminus a) = \mathtt{fv}(P)$, $\mathtt{fv}(P + Q) = \mathtt{fv}(P \mid Q) = \mathtt{fv}(P) \cup \mathtt{fv}(Q)$, $\mathtt{fv}(\mathsf{rec}\ x.\, P) = \mathtt{fv}(P) \setminus \{x\}$, and $\mathtt{fv}(x) = \{x\}$. A process P is closed if $\mathtt{fv}(P) = \emptyset$, and is guarded if any variable appears underneath a prefix a, $\overline{a}$, and τ. In what follows we assume closed and guarded processes.

Definition 5.9 (Process convergence) *Let* $\Downarrow$, *read* P converges, *be the least predicate over processes such that:*

$$
\begin{array}{ll}
\mathbf{0} \Downarrow & \\
\alpha.P \Downarrow & \\
\tau.P \Downarrow & \text{if } P \Downarrow \\
P \setminus a \Downarrow & \text{if } P \Downarrow \\
P + Q \Downarrow & \text{if } P \Downarrow \text{ and } Q \Downarrow \\
P \mid Q \Downarrow & \text{if } P \Downarrow \text{ and } Q \Downarrow \text{ and } P \xrightarrow{\alpha} P',\ Q \xrightarrow{\overline{\alpha}} Q' \text{ implies } P' \mid Q' \Downarrow \\
\mathsf{rec}\ x.\, P \Downarrow & \text{if } P\{^{\mathsf{rec}\ x.\,P}/_{x}\} \Downarrow
\end{array}
$$

We write $P \not\Downarrow$ *if not* $P \Downarrow$.

For instance $\mathsf{rec}\ x.\, \tau.x + a \not\Downarrow$ and $\mathsf{rec}\ x.\, a.x \mid \mathsf{rec}\ y.\, \overline{a}y \not\Downarrow$.

Definition 5.10 (Process ready set) *Let* R *range over finite sets of names and co-names, called* ready sets. *Let* $P \searrow \mathrm{R}$, *ready* P has ready set R, *be the least relation such that:*

$$
\begin{array}{rl}
\mathbf{0} \searrow \emptyset & \\
\alpha.P \searrow \{\alpha\} & \\
\tau.P \searrow \mathrm{R} & \text{if } P \searrow \mathrm{R} \\
P \setminus a \searrow \mathrm{R} \setminus \{a, \overline{a}\} & \text{if } P \searrow \mathrm{R} \\
P + Q \searrow \mathrm{R}_1 \cup \mathrm{R}_2 & P \searrow \mathrm{R}_1 \text{ and } Q \searrow \mathrm{R}_2 \\
P + Q \searrow \mathrm{R} & \text{if either } P \xrightarrow{\tau} P' \searrow \mathrm{R} \text{ or } Q \xrightarrow{\tau} Q' \searrow \mathrm{R} \\
P \mid Q \searrow \mathrm{R} & \text{if } P \xrightarrow{\alpha} P',\ Q \xrightarrow{\overline{\alpha}} Q' \text{ and } P' \mid Q' \searrow \mathrm{R} \\
P \mid Q \searrow \mathrm{R}_1 \cup \mathrm{R}_2 \cup \mathrm{R}_3 & \textit{if } P \searrow \mathrm{R}_1 \text{ and } Q \searrow \mathrm{R}_2 \text{ and } P \xrightarrow{\alpha} P',\ Q \xrightarrow{\overline{\alpha}} Q',\ P' \mid Q' \searrow \mathrm{R}_3 \\
P \mid Q \searrow \mathrm{R}_1 \cup \mathrm{R}_2 & \textit{if } P \searrow \mathrm{R}_1 \text{ and } Q \searrow \mathrm{R}_2 \text{ and } P \xrightarrow{\alpha} \text{ implies } Q \overset{\overline{\alpha}}{\not\longrightarrow} \\
\mathsf{rec}\ x.\, P \searrow \mathrm{R} & \text{if } P\{^{\mathsf{rec}\ x.\,P}/_{x}\} \searrow \mathrm{R} \\
P \searrow \emptyset & \text{if } P \not\Downarrow
\end{array}
$$

As in the definition of ready set of contracts, the ready sets of a process P collect set of visible actions where the process may be waiting for the interacting party. For instance the process $\tau.a + \tau.b$ has three ready sets $\{a\}$, $\{b\}$ and $\{a, b\}$. Indeed in the initial state both a and b are potentially available, however, by means of internal moves, it may reduce to either a or b. In case of process with synchronizations we have to take into account the

ready set of the continuation. Then $a.b \mid \overline{a}$ has $\{b\}$ and $\{a, \overline{a}, b\}$ as ready sets. Indeed the process may internally choose to perform the synchronization, thus becoming available on b, or to wait for the interacting party in the initial state $a.b \mid \overline{a}$. In this second case, we note that the process is also ready b (not only on a, $\overline{a}$) because it keeps the possibility of showing b to interacting parties. To strengthen this aspects, we remark that the process $\overline{b}$ interacts successfully with $a.b \mid \overline{a}$, then the name b is always available (if a branch does not show b then $\overline{b}$ would deadlock). As regards ready sets for recursive process, we note that $(\mathsf{rec}\ x.\, a.x) \mid (\mathsf{rec}\ x.\, \overline{a}.x)$ has $\{a, \overline{a}\}$ and $\emptyset$ as unique ready sets. The presence of the ready set $\emptyset$ is because the process may diverge.

We remark that Definition 5.9 and Definition 5.10 are undecidable for non-regular processes [BGZ03, BGZ04]. In particular the undecidability of Definition 5.10 follows by the undecidability of the property "*P has barb α*" for CCS with recursion (the property is decidable in CCS with replication [BGZ04]). On the contrary, both the definitions are decidable for regular processes because their evaluation amounts to verify the finitely many states of the process.

Proposition 5.9 *If $P \stackrel{\tau}{\longrightarrow} P'$ then $P' \searrow \mathrm{R}$ implies $P \searrow \mathrm{R}$*

Proof: We proceed by induction on the derivation of $P \stackrel{\tau}{\longrightarrow}$. The base case is $P = \tau.Q$ and is immediate, P and Q have the same ready sets. The inductive cases are:

$P = Q \setminus a$ and $Q \stackrel{\tau}{\longrightarrow} Q'$. By the inductive hypothesis, $Q' \searrow \mathrm{R}$ implies $Q \searrow \mathrm{R}$ and, by definition of ready set, if $Q' \searrow \mathrm{R}$ and $Q \searrow \mathrm{R}$ then $Q' \setminus a \searrow \mathrm{R} \setminus \{a, \overline{a}\}$ and $Q \setminus a \searrow \mathrm{R} \setminus \{a, \overline{a}\}$. We conclude $Q' \setminus a \searrow \mathrm{R}$ implies $Q \setminus a \searrow \mathrm{R}$.

$P = Q \mid Q'$ and $Q \stackrel{\tau}{\longrightarrow} Q''$. By the inductive hypothesis, $Q'' \searrow \mathrm{R}$ implies $Q \searrow \mathrm{R}$ By definition of ready set of parallel, we conclude $Q \mid Q'' \searrow \mathrm{R}$ implies $Q' \mid Q'' \searrow \mathrm{R}$.

$P = \mathsf{rec}\ x.\, Q$ and $Q\{^P/_x\} \stackrel{\tau}{\longrightarrow} Q'$. By the inductive hypothesis, $Q' \searrow \mathrm{R}$ implies $Q\{^P/_x\} \searrow \mathrm{R}$ then, by definition of ready sets P and $Q\{^P/_x\}$ have the same ready sets. Thus we conclude $Q' \searrow \mathrm{R}$ implies $P \searrow \mathrm{R}$.

$P = Q + Q'$ and $Q \stackrel{\tau}{\longrightarrow} Q''$. Since $Q' \searrow \mathrm{R}$ implies $Q \searrow \mathrm{R}$ then, by definition of ready sets, $Q' + Q'' \searrow \mathrm{R}$ implies $P \searrow \mathrm{R}$. $\square$

We now formally define when a process P complies with a process Q.

Definition 5.11 (Compliance) *Let $P \parallel Q \longrightarrow P' \parallel Q'$ be the least relation such that:*

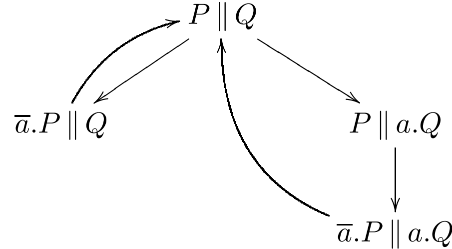
- *if $P \xrightarrow{\tau} P'$ then $P \parallel Q \longrightarrow P' \parallel Q$;*
- *if $Q \xrightarrow{\tau} Q'$ then $P \parallel Q \longrightarrow P \parallel Q'$;*
- *if $P \xrightarrow{\alpha} P'$ and $Q \xrightarrow{\overline{\alpha}} Q'$ then $P \parallel Q \longrightarrow P' \parallel Q'$.*

$\mathcal{C}$ is a compliance relation if and only if $P\mathcal{C}Q$ implies

1. *either $P \stackrel{\mu}{\not\longrightarrow}$,*
2. *(a) $P \stackrel{\tau}{\not\longrightarrow}$ implies $Q \Downarrow$,*
   (b) and $P \parallel Q \longrightarrow P' \parallel Q'$ with $P'\mathcal{C}Q'$.

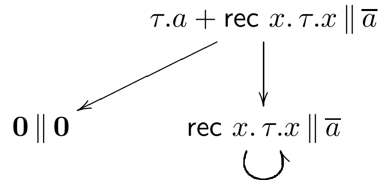
Let $\between_p$ be the largest compliance relation. We say that P complies with *Q if $P \between_p Q$.*

Process compliance extends the definition of Chapter 4 with clause 2.a. According to the new definition, P complies with Q if either P is inactive or for every transition of $P \parallel Q \longrightarrow P' \parallel Q'$, P' complies with Q'. We note that condition 2.a together with 2.b verifies that, whenever P needs to interact on some name, Q must offer such a name.
Example: Let $P = \mathsf{rec}\ x.\,\tau.\overline{a}.x + \overline{b}.x$ and $Q = \mathsf{rec}\ x.\,a.x + b.a.x$. $P \between_p Q$. As regards transitions we have:



It is easy to see that $\mathcal{C} = \{(P,Q),\ (\overline{a}.P,Q),\ (P,a.Q),\ (\overline{a}.P,a.Q)\}$ is a compliance relation.

Example: $\tau.a + \mathsf{rec}\ x.\,\tau.x \between_p \overline{a}$. As regards transitions we have:



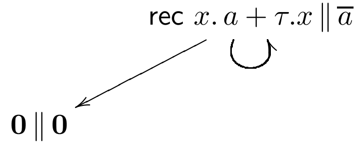
We choose $\mathcal{C} = \{(\tau.a + \mathsf{rec}\ x.\,\tau.x, \overline{a}),\ (\mathbf{0},\mathbf{0}),\ (\mathsf{rec}\ x.\,\tau.x, \overline{a})\}$ as compliance relation.

Example: For every Q, $a + \mathsf{rec}\ x.\, \tau.x \between_p Q$. Let

$$\mathcal{C} = \{(a + \mathsf{rec}\ x.\, \tau.x, Q),\ (\mathbf{0}, P),\ (\mathsf{rec}\ x.\, \tau.x, R) : Q, P, R \text{ are processes}\}$$

We have either: (a) $a + \mathsf{rec}\ x.\, \tau.x \parallel Q \longrightarrow \mathbf{0} \parallel P$ with $(\mathbf{0}, P) \in \mathcal{C}$ or (b) $a + \mathsf{rec}\ x.\, \tau.x \parallel Q \longrightarrow \mathsf{rec}\ x.\, \tau.x \parallel R$ with $(\mathsf{rec}\ x.\, \tau.x, R) \in \mathcal{C}$. We note that 2.a of Definition 5.11 is always verified because the premise fails.

Example: $\mathsf{rec}\ x.\, a + \tau.x \between_p \overline{a}$



It is easy to verify that $\mathcal{C} = \{(\mathsf{rec}\ x.\, a + \tau.x, \overline{a}),\ (\mathbf{0}, \mathbf{0})\}$ is a compliance relation. We also note that $\overline{a} \not\between_p \mathsf{rec}\ x.\, a + \tau.x$ because, by condition 2.a of Definition 5.11, $\overline{a} \stackrel{a}{\longrightarrow}$, $\overline{a} \stackrel{\tau}{\not\longrightarrow}$ and $\mathsf{rec}\ x.\, a + \tau.x \not\Downarrow$.

Proposition 5.10 *If $P \between_p Q$ and $P \stackrel{\alpha}{\Longrightarrow} P'$ and $Q \stackrel{\overline{\alpha}}{\Longrightarrow} Q'$ then $P' \between_p Q'$*

Proof: Follows by induction on the derivation of $P \stackrel{\alpha}{\Longrightarrow}$ and $Q \stackrel{\overline{\alpha}}{\Longrightarrow}$. □

5.4 Process Estimations

5.4.1 Contracts and processes

A contract is an abstract description of the communication protocol implemented by a process. Of course, it is possible to abstract away from protocols details at wish. However the abstraction process cannot be unconstrained in our case. Indeed, when a contract σ is derived from a server Q we must verify that every client compliant with the process described by σ (the canonical implementation of σ) is also compliant with Q. Symmetrically, when a contract σ is derived from a client P we must verify that every server compliant with the process described by σ is also compliant with P.

To introduce the definition we note that, when a contract has exactly one ready set, it leaves to the interacting party the possibility of choosing the name for the communication. When a contract has several ready sets then – after an internal choice – the set of names

in one of these ready sets is available for communications to the interacting parties. For instance, a contract having only one ready set $\{a, b\}$ is implemented by the process $a + b$. On the contrary, a contract having ready sets $\{a, b\}$ and $\{a, c\}$ is implemented by the process $\tau.(a + b) + \tau.(a + c)$. The following definition formalizes this concept.

Definition 5.12 *Let* $\mathtt{R}(\sigma) = \bigcup_{\sigma \searrow \mathrm{R}} \{\mathrm{R}\}$. *The process* $\mathtt{rt}(\sigma)$ *is defined as* $\mathtt{rt}_{\emptyset}(\sigma)$ *where*

$$\mathtt{rt}_A(\sigma) \stackrel{\text{def}}{=} \begin{cases} \sum_{\alpha\in\mathrm{R}} \alpha.\mathtt{rt}_{A'}(\sigma(\alpha)) & \text{if } \sigma \notin \mathtt{dom}(A),\ \mathtt{R}(\sigma) = \{\mathrm{R}\}, \\ & A' = A + [\sigma \mapsto x],\ x \notin \mathtt{fv}(\mathtt{rt}_{A'}(\sigma(\alpha))) \\ \mathsf{rec}\ x.\ \sum_{\alpha\in\mathrm{R}} \alpha.\mathtt{rt}_{A'}(\sigma(\alpha)) & \text{if } \sigma \notin \mathtt{dom}(A),\ \mathtt{R}(\sigma) = \{\mathrm{R}\}, \\ & A' = A + [\sigma \mapsto x],\ x \in \mathtt{fv}(\mathtt{rt}_{A'}(\sigma(\alpha))) \\ \sum_{\mathrm{R}\in\mathtt{R}(\sigma)} \tau.\sum_{\alpha\in\mathrm{R}} \alpha.\mathtt{rt}_{A'}(\sigma(\alpha)) & \text{if } \sigma \notin \mathtt{dom}(A),\ \mid \mathtt{R}(\sigma) \mid > 1, \\ & A' = A + [\sigma \mapsto x],\ x \notin \mathtt{fv}(\mathtt{rt}_{A'}(\sigma(\alpha))) \\ \mathsf{rec}\ x.\ \sum_{\mathrm{R}\in\mathtt{R}(\sigma)} \tau.\sum_{\alpha\in\mathrm{R}} \alpha.\mathtt{rt}_{A'}(\sigma(\alpha)) & \text{if } \sigma \notin \mathtt{dom}(A),\ \mid \mathtt{R}(\sigma) \mid > 1, \\ & A' = A + [\sigma \mapsto x],\ x \in \mathtt{fv}(\mathtt{rt}_{A'}(\sigma(\alpha))) \\ A(\sigma) & \text{if } \sigma \in \mathtt{dom}(A) \end{cases}$$

The definition of $\mathtt{rt}_A(\sigma)$ distinguishes five cases. The last case is for closing the contract in case of recursion (we remind that in $A' = A + [\sigma \mapsto x]$ $x \notin \mathtt{cod}(A)$). The first two cases correspond to contracts manifesting exactly one ready set (there are two different cases because recursive contracts are created only where needed). For instance, since $\mathtt{R}(\mathbf{\Omega}) = \{\emptyset\}$, we have $\mathtt{rt}(\mathbf{\Omega}) = \tau.\mathbf{0}$. Similarly, since $\mathtt{R}(\mathsf{rec}\ x.\, a.x + b.x) = \{a, b\}$, we have $\mathtt{rt}(\mathsf{rec}\ x.\, a.x + b.x) = \mathsf{rec}\ x.\, a.x + b.x$. The third and the fourth cases correspond to contracts manifesting more than one ready set. In such cases a τ move – implementing the internal choice among ready sets – is prefixed to continuation. For instance, since $\mathtt{R}(a \oplus b) = \{\{a\},\ \{b\}\}$ and $(a \oplus b)(a) = (a \oplus b)(b) = \mathbf{0}$, we have $\mathtt{rt}(a \oplus b) = \tau.a + \tau.b$. Some other examples are:

1. $\mathtt{rt}(c.a \oplus c.b) = c.(\tau.a + \tau.b)$; indeed $\mathtt{R}(c.a \oplus c.b) = \{\{c\}\}$ and $(c.a \oplus c.b)(c) = (a \oplus b)$

2. $\mathtt{rt}(\mathsf{rec}\ x.\, a + x) = \mathtt{rt}(\mathsf{rec}\ x.\, a \oplus x) = \mathtt{rt}(a \oplus \mathbf{0}) = \tau.a + \tau.\mathbf{0}$; indeed $\mathtt{R}(\mathsf{rec}\ x.\, a + x) = \mathtt{R}(\mathsf{rec}\ x.\, a \oplus x) = \mathtt{R}(a \oplus \mathbf{0}) = \{\{a\},\ \emptyset\}$

3. Let $\sigma = \mathsf{rec}\ x.\, a.x \oplus \mathsf{rec}\ y.\, a.b.y$. $\mathtt{rt}(\sigma) = a.(\tau.a.\mathsf{rec}\ x.\, a.x + \tau.b.\mathsf{rec}\ y.\, a.b.y)$. Indeed $\mathtt{R}(\sigma) = \{\{a\}\}$, $(\sigma)(a) = \sigma \oplus b.\mathsf{rec}\ y.\, .a.b.y$ and $\mathtt{R}(\sigma \oplus b.\mathsf{rec}\ y.\, .a.b.y) = \{\{a\},\{b\}\}$, $(\sigma \oplus b.\mathsf{rec}\ y.\, a.b.y)(a) = \sigma$, and $(\sigma \oplus b.\mathsf{rec}\ y.\, a.b.y)(b) = \mathsf{rec}\ y.\, a.b.y$.

We note that the definition may be simplified creating a recursive definition $\mathsf{rec}\ x.\, _$ at every step. Moreover, we also remark that the naïve definition by induction on the structure of the contract: $\mathtt{rt}(\mathbf{0}) = \mathbf{0}$, $\mathtt{rt}(\alpha.\sigma) = \alpha.\mathtt{rt}(\sigma)$, $\mathtt{rt}(\sigma + \sigma') = \mathtt{rt}(\sigma) + \mathtt{rt}(\sigma')$, and $\mathtt{rt}(\sigma \oplus \sigma') = \tau.\mathtt{rt}(\sigma) + \tau.\mathtt{rt}(\sigma')$, ... is fallible. Indeed, from $(a \oplus b) + c$ we obtain $\tau.a + \tau.b + c$ which does not implement the protocol described by the contract.

In order to show the strong relation between σ and $\mathtt{rt}(\sigma)$ we need the following preliminary Proposition.

Proposition 5.11 $\mathtt{rt}_{A+[\sigma\mapsto x]}(\sigma')\{^{\mathtt{rt}_A(\sigma)}/_x\} = \mathtt{rt}_A(\sigma')$

Proof: We proceed by induction on the structure of $\mathtt{rt}_{A+[\sigma\mapsto x]}(\sigma')$. The case $\mathtt{rt}_{A+[\sigma\mapsto x]}(\sigma') = \mathbf{0}$ is immediate. In case of $\mathtt{rt}_{A+[\sigma\mapsto x]}(\sigma') = x$, by definition of $\mathtt{rt}_A(\cdot)$, $\sigma' = \sigma$ and we conclude. In case of $\mathtt{rt}_{A+[\sigma\mapsto x]}(\sigma') = y \neq x$, by definition of $\mathtt{rt}_A(\cdot)$, $A(\sigma') = y$ and $\sigma' \neq \sigma$. Then $\mathtt{rt}_{A+[\sigma\mapsto x]}(\sigma')\{^{\mathtt{rt}_A(\sigma)}/_x\} = y$ and $\mathtt{rt}_A(\sigma') = y$. Hence we conclude. As regards the inductive cases we distinguish four cases depending on which rule of $\mathtt{rt}_{A+[\sigma\mapsto x]}$ has been applied. We discuss those having $\mathtt{R}(\sigma) = \{\mathtt{R}\}$, other two being similar.

- If $\mathtt{rt}_{A+[\sigma\mapsto x]}(\sigma') = P$ with $P = \sum_{\alpha\in\mathtt{R}} \alpha.R_\alpha$ with $R_\alpha = \mathtt{rt}_{A+[\sigma\mapsto x]+[\sigma'\mapsto y]}(\sigma'(\alpha))$ and $y \notin \mathtt{fv}(R_\alpha)$. By the inductive hypothesis $R_\alpha\{^{\mathtt{rt}_A(\sigma)}/_x\} = \mathtt{rt}_{A+[\sigma'\mapsto y]}(\sigma'(\alpha))$. By $\sigma' \notin \mathtt{dom}(A)$, $y \notin \mathtt{fv}(\mathtt{rt}_A(\sigma))$ and $y \notin \mathtt{fv}(R_\alpha\{^{\mathtt{rt}_A(\sigma)}/_x\})$. Hence $y \notin \mathtt{fv}(\mathtt{rt}_{A+[\sigma'\mapsto y]}(\sigma'(\alpha)))$ and, by definition of $\mathtt{rt}(\cdot)$, $\mathtt{rt}_A(\sigma') = \sum_{\alpha\in\mathtt{R}} \alpha.\mathtt{rt}_{A+[\sigma'\mapsto y]}(\sigma'(\alpha))$. We conclude by the inductive hypothesis.
- If $\mathtt{rt}_{A+[\sigma\mapsto x]}(\sigma') = \mathsf{rec}\ y.\, \sum_{\alpha\in\mathtt{R}} \alpha.R_\alpha$ with $R_\alpha = \mathtt{rt}_{A+[\sigma\mapsto x]+[\sigma'\mapsto y]}(\sigma'(\alpha))$ and $y \in \mathtt{fv}(R_\alpha)$. By the inductive hypothesis $R_\alpha\{^{\mathtt{rt}_A(\sigma)}/_x\} = \mathtt{rt}_{A+[\sigma'\mapsto y]}(\sigma'(\alpha))$. By $y \in \mathtt{fv}(R_\alpha)$, $y \in \mathtt{fv}(R_\alpha\{^{\mathtt{rt}_A(\sigma)}/_x\})$ and $y \in \mathtt{fv}(\mathtt{rt}_{A+[\sigma'\mapsto y]}(\sigma'(\alpha)))$. Hence, by definition of $\mathtt{rt}(\cdot)$, $\mathtt{rt}_A(\sigma') = \mathsf{rec}\ y.\, \sum_{\alpha\in\mathtt{R}} \alpha.\mathtt{rt}_{A+[\sigma'\mapsto y]}(\sigma'(\alpha))$. We conclude by the inductive hypothesis. □

Proposition 5.12

1. $\mathtt{rt}(\sigma)\Downarrow$;

2. *if* $\mathtt{rt}(\sigma) \xrightarrow{\alpha} Q$ *then* $Q = \mathtt{rt}(\sigma(\alpha))$;

3. *if* $\mathtt{rt}(\sigma) \xrightarrow{\tau} Q$ *then* $\sigma \searrow \mathrm{R}$ *and* $Q = \sum_{\alpha \in \mathrm{R}} \alpha.\mathtt{rt}(\sigma(\alpha))$;

4. *if* $\sigma \overset{\alpha}{\longmapsto} \sigma(\alpha)$ *then* $\mathtt{rt}(\sigma) \overset{\alpha}{\Longrightarrow} \mathtt{rt}(\sigma(\alpha))$.

Proof: Item 1 is immediate because, by definitions of $\mathtt{rt}_A(\sigma)$, contracts names are always under a prefix α.

Item 2 and item 3 are similar. In case of item 2 we proceed by induction on the derivation of $\mathtt{rt}(\sigma) \xrightarrow{\alpha} Q$. The base case is when $\mathtt{rt}(\sigma)$ is due to (ACT). Then $\mathtt{rt}(\sigma) = \alpha.P$ and $P = \mathtt{rt}_{[\sigma \mapsto x]}\sigma(\alpha)$. In this case $x \notin \mathtt{fv}(\mathtt{rt}_{[\sigma \mapsto x]}\sigma(\alpha))$ therefore $\mathtt{rt}_{[\sigma \mapsto x]}\sigma(\alpha) = \mathtt{rt}(\sigma(\alpha))$ and we conclude. In the inductive case we have two cases.

- The last rule used in the derivation of $\mathtt{rt}(\sigma) \xrightarrow{\alpha} P''$ is a (CHOICE). Then $\mathtt{rt}(\sigma) = P + P'$ and, by definition of $\mathtt{rt}(\sigma)$, $P + P' = \sum_{\alpha \in \mathrm{R}} \alpha.\mathtt{rt}_{[\sigma \mapsto x]}(\sigma(\alpha))$, $x \notin \mathtt{fv}(\mathtt{rt}_{[\sigma \mapsto x]}\sigma(\alpha))$. Hence we conclude by the inductive hypothesis $P'' = \mathtt{rt}(\sigma(\alpha))$.

- The last rule used in the derivation of $\mathtt{rt}(\sigma) \xrightarrow{\alpha} P''$ is a (REC). Then $\mathtt{rt}(\sigma) = \mathsf{rec}\ x.\,P \xrightarrow{\alpha} P'$ and the premise of (REC) is $P\{^{\mathsf{rec}\ x.\,P}/_x\} \xrightarrow{\alpha} P'$. By definition of $\mathtt{rt}(\cdot)$, $P = \sum_{\alpha \in \mathrm{R}} \alpha.\mathtt{rt}_{[\sigma \mapsto x]}\sigma(\alpha)$. Thus $P\{^{\mathsf{rec}\ x.\,P}/_x\} = (\sum_{\alpha \in \mathrm{R}} \alpha.\mathtt{rt}_{[\sigma \mapsto x]}\sigma(\alpha))\{^{\mathtt{rt}(\sigma)}/_x\}$ that is, by Proposition 5.11, $\sum_{\alpha \in \mathrm{R}} \alpha.\mathtt{rt}(\sigma(\alpha))$. Thus we conclude.

As regards item 4, the hypothesis $\sigma \overset{\alpha}{\longmapsto}$ implies $\sigma \searrow \mathrm{R}$ and $\alpha \in \mathrm{R}$. Then, by definition of $\mathtt{rt}(\sigma)$, either $\mathtt{R}(\sigma) = \{\mathrm{R}\}$ or $\mid \mathtt{R}(\sigma) \mid > 1$. In the first case, by item 2 and by definition of $\mathtt{rt}(\cdot)$, $\mathtt{rt}(\sigma) \xrightarrow{\alpha} \mathtt{rt}(\sigma(\alpha))$. In the second case, by item 3 and by definition of $\mathtt{rt}(\cdot)$, $\mathtt{rt}(\sigma) \xrightarrow{\tau}\xrightarrow{\alpha} \mathtt{rt}(\sigma(\alpha))$. $\square$

Lemma 5.4 *If* $\sigma \between_c \sigma'$ *then* $\mathtt{rt}(\sigma) \between_p \mathtt{rt}(\sigma')$.

Proof: Let

$$\mathcal{R} = \{(\mathtt{rt}(\sigma), \mathtt{rt}(\sigma')) : \overline{\sigma} \preceq \sigma'\}.$$

We show that $\mathcal{R}$ is a process compliance relation. We distinguish two cases. If $\mathtt{rt}(\sigma) \overset{\mu}{\not\longrightarrow}$ then we immediately conclude. If $\mathtt{rt}(\sigma) \xrightarrow{\mu}$ then we have to prove that (a) $\mathtt{rt}(\sigma) \overset{\tau}{\not\longrightarrow}$

implies $\mathtt{rt}(\sigma') \Downarrow$ and (b) $\mathtt{rt}(\sigma) \parallel \mathtt{rt}(\sigma') \longrightarrow P \parallel Q$ with $(P, Q) \in \mathcal{R}$. By the hypothesis $\overline{\sigma} \preceq \sigma'$ we have:

$$\sigma' \searrow \mathrm{R}' \text{ implies } \overline{\sigma} \searrow \mathrm{R} \subseteq \mathrm{R}' \tag{5.1}$$

$$\overline{\sigma} \stackrel{\alpha}{\longmapsto} \overline{\sigma(\alpha)},\ \sigma' \stackrel{\alpha}{\longmapsto} \sigma'(\alpha) \text{ implies } \overline{\sigma(\alpha)} \preceq \sigma'(\alpha) \tag{5.2}$$

As regards (a), by Proposition 5.12 $\mathtt{rt}(\sigma') \Downarrow$.

As regards (b) there are several cases depending on the transition of $\mathtt{rt}(\sigma) \parallel \mathtt{rt}(\sigma')$.

$\mathtt{rt}(\sigma) \stackrel{\tau}{\longrightarrow} Q$. By Proposition 5.12, $Q = \sum_{\alpha \in \mathrm{R}} \alpha.\mathtt{rt}(\sigma(\alpha))$ and $\sigma \searrow \mathrm{R}$. By Definition 5.12, $\sum_{\alpha \in \mathrm{R}} \alpha.\mathtt{rt}(\sigma(\alpha)) = \mathtt{rt}(\sum_{\alpha \in \mathrm{R}} \alpha.\sigma(\alpha))$. Therefore, we are reduced to demonstrate that $(Q, \mathtt{rt}(\sigma')) \in \mathcal{R}$, that is $\overline{\sum_{\alpha \in \mathrm{R}} \alpha.\sigma(\alpha)} \preceq \sigma'$. By Proposition 5.6.2, for every $\alpha \in \mathrm{R}$, $\overline{\sum_{\alpha \in \mathrm{R}} \alpha.\sigma(\alpha)} \searrow \{\overline{\alpha}\}$ and $\overline{\sigma} \searrow \mathrm{R}$ with $\{\overline{\alpha}\} \subseteq \mathrm{R}$. By (5.1) and transitivity of $\subseteq$, condition 1 of $\preceq$ is verified. By (5.2), condition 2 of $\preceq$ is also verified. Hence we conclude $\overline{\sum_{\alpha \in \mathrm{R}} \sigma(\alpha)} \preceq \sigma'$.

$\mathtt{rt}(\sigma') \stackrel{\tau}{\longrightarrow} Q$. By Proposition 5.12, $Q = \sum_{\alpha \in \mathrm{R}} \alpha.\mathtt{rt}(\sigma(\alpha))$ and $\sigma \searrow \mathrm{R}$. By Definition 5.12, $\sum_{\alpha \in \mathrm{R}} \alpha.\mathtt{rt}(\sigma(\alpha)) = \mathtt{rt}(\sum_{\alpha \in \mathrm{R}} \alpha.\sigma(\alpha))$. Therefore we are reduced to demonstrate $(\mathtt{rt}(\sigma), Q) \in \mathcal{R}$, that is $\overline{\sigma} \preceq \sum_{\alpha \in \mathrm{R}} \alpha.\sigma(\alpha)$. By definition of ready set $\sum_{\alpha \in \mathrm{R}} \alpha.\sigma'(\alpha) \searrow \mathrm{R}''$ implies $\sigma' \searrow \mathrm{R}''$. Thus, by (5.1), condition 1 of $\preceq$ is verified. By (5.2), condition 2 of $\preceq$ is also verified. Hence we conclude $\overline{\sigma} \preceq \sum_{\alpha \in \mathrm{R}} \sigma'(\alpha)$.

$\mathtt{rt}(\sigma) \stackrel{\alpha}{\longrightarrow} P, \mathtt{rt}(\sigma') \stackrel{\overline{\alpha}}{\longrightarrow} Q$. By Proposition 5.12, $P = \mathtt{rt}(\sigma(\alpha))$ and $Q = \mathtt{rt}(\sigma'(\overline{\alpha}))$. Then we are reduced to prove $(\mathtt{rt}(\sigma(\alpha)), \mathtt{rt}(\sigma'(\overline{\alpha}))) \in \mathcal{R}$, that is $\sigma(\alpha) \preceq \sigma'(\overline{\alpha})$. We conclude by $\overline{\sigma} \preceq \sigma'$ and Proposition 5.7. □

5.4.2 Underestimation

Underestimation is used for assigning proper contracts to service providers. Intuitively a contract underestimating a process describes a protocol such that it is more difficult, for clients, to be compliant with. In this section we first define the underestimation in a set theoretic way and then we give an operational definition – a simulation – for verifying whenever a contract underestimates the protocol implemented by a processes. Thus we prove that the operational definition implies the set-theoretic one.

Definition 5.13 (Set-Theoretic Underestimation) *We say that* σ underestimates Q *if and only if* $\{Q : Q \between_p \mathtt{rt}(\sigma)\} \subseteq \{Q : Q \between_p P\}$.

Definition 5.14 (Operational Underestimation) *A relation over processes* $\mathcal{U}$ *is an underestimation relation if* $P\mathcal{U}\sigma$ *implies*

1. *either* $\sigma \searrow \emptyset$
2. *or the following conditions hold:*
   (a) *if* $P \searrow \mathrm{R}$ *then* $\sigma \searrow \mathrm{R}'$ *with* $\mathrm{R}' \subseteq \mathrm{R}$
   (b) *if* $P \stackrel{\alpha}{\Longrightarrow} P'$ *then* $\sigma \stackrel{\alpha}{\longmapsto} \sigma'$ *with* $P'\mathcal{U}\sigma'$

Let $\vdash_u$ *be the largest underestimation relation.*

A contract σ underestimating a process P describes the protocol implemented by P in more restricting way for clients. At first we note that a contract with an empty ready sets underestimates any process. Indeed it means that the server may decide to do not interact at all. Thus client's protocols compliant with such a server cannot interact. If σ does not have empty ready sets the contract must be less deterministic. Indeed, item 1 says that for any ready set of P – corresponding to choices left open to the interacting party – there is a corresponding external choice on a smaller set of names. Item 2 deals with transitions. If $P \stackrel{\alpha}{\Longrightarrow} P'$ we require σ to perform the same action with a successor σ' that underestimates P' (for every P'). For instance, we may verify that $a + b \vdash_u a \oplus b$, and $a.c + a.d + b \vdash_u a + b$. On the other hand, $a + b \not\vdash_u a$ (because of the item 2.b) and $a \not\vdash_u a \oplus b$ because of the item 2.a. Indeed, the process $\overline{a} + \overline{b}.c$ terminates interacting with a but not with $a+b$. We notice that, by item 2 we have that $\mathtt{traces}(P) \subseteq \mathtt{traces}(\sigma)$ and by item 1 it is always possible to approximate with a contract having and empty ready set. It follows that, until a certain depth n (the one having $\sigma \searrow \emptyset$), traces of σ include traces of P.

Proposition 5.13

1. *for every* P, $P \vdash_u \mathbf{\Omega}$ *and* $P \vdash_u \mathbf{0}$
2. $\alpha \vdash_u \alpha.P$

Proposition 5.14 *If* $P \vdash_u \sigma$ *and* $P \xrightarrow{\tau} P'$ *then* $P' \vdash_u \sigma$

Proof: By Proposition 5.9 $P \xrightarrow{\tau} P' \searrow \mathrm{R}$ implies $P \searrow \mathrm{R}$. Moreover, if $P' \stackrel{\alpha}{\Longrightarrow} P''$ then $P \stackrel{\alpha}{\Longrightarrow} P''$. Therefore we conclude by the hypothesis $P \vdash_u \sigma$. □

Lemma 5.5 $P \vdash_u \sigma$ *then* σ underestimates P.

Proof: We have to show that if $P \vdash_u \sigma$ then $Q \between_p \mathtt{rt}(\sigma)$ implies $Q \between_p P$ (that is $\{Q : Q \between_p \mathtt{rt}(\sigma)\} \subseteq \{Q : Q \between_p P\}$). This is equivalent to show that

$$\mathcal{R} \quad = \quad \{(Q, P) : P \vdash_u \sigma,\ Q \between_p \mathtt{rt}(\sigma)\}$$

is a compliance relation. By the hypothesis $P \vdash_u \sigma$ we have two cases. If $\sigma \searrow \emptyset$ then $Q \between_p \mathtt{rt}(\sigma)$ implies $Q \stackrel{\alpha}{\not\Longrightarrow}$ and we conclude. Otherwise we have:

$$P \searrow \mathrm{R} \text{ implies } \sigma \searrow \mathrm{R}' \text{ with } \mathrm{R}' \subseteq \mathrm{R} \tag{5.3}$$

$$P \stackrel{\alpha}{\Longrightarrow} P' \text{ implies } \sigma \stackrel{\alpha}{\longmapsto} \sigma' \text{ with } P' \vdash_u \sigma' \tag{5.4}$$

We distinguish two cases (a) $Q \stackrel{\mu}{\not\longrightarrow}$ and (b) $Q \xrightarrow{\mu}$. In case (a) we conclude. In case (b) we must show (b1) $Q \xrightarrow{\alpha}$ and $Q \stackrel{\tau}{\not\longrightarrow}$ implies $P \Downarrow$, and (b2) $Q \parallel P \longrightarrow Q' \parallel P'$ implies $(Q', P') \in \mathcal{R}$.

As regards (b1), let assume $P \not\Downarrow$. Then $P \searrow \emptyset$ and, by (5.3) $\sigma \searrow \emptyset$. By definition of $\mathtt{rt}(\sigma)$, either $\mathtt{rt}(\sigma) = \mathbf{0}$ or $\mathtt{rt}(\sigma) = \tau.\mathbf{0}$. Hence, since we have $Q \xrightarrow{\alpha}$, we conclude $Q \not\between_c \mathtt{rt}(\sigma)$ which is absurd. Then P converges.

As regards (b2), we distinguish three subcases:

$Q \xrightarrow{\tau} Q'$ By $Q \between_p \mathtt{rt}(\sigma)$ we have $Q' \between_p \mathtt{rt}(\sigma)$. Thus we conclude $(P, Q') \in \mathcal{R}$.

$P \xrightarrow{\tau} P'$ By Proposition 5.14 we have $P' \vdash_u \sigma$. Thus we conclude $(P', Q) \in \mathcal{R}$.

$Q \xrightarrow{\alpha} Q'$**,** $P \xrightarrow{\overline{\alpha}} P'$ By (5.4) we have that $\sigma \stackrel{\overline{\alpha}}{\longmapsto} \sigma(\overline{\alpha})$ with $P' \vdash_u \sigma(\overline{\alpha})$. By Proposition 5.12, $\mathtt{rt}(\sigma) \stackrel{\overline{\alpha}}{\Longrightarrow} \mathtt{rt}(\sigma(\overline{\alpha}))$, and, by Proposition 5.10, $Q' \between_p \mathtt{rt}(\sigma(\overline{\alpha}))$. Thus we conclude $(Q', P') \in \mathcal{R}$. □

The following example show how a non-regular process can be underestimated.

Example: Let $P = \mathsf{rec}\ x. (a.x \mid b)$ and $\sigma = \mathsf{rec}\ x. a.x \oplus b.x$ then $P \vdash_u \sigma$.

Proof: In this example we reason up to a structural congruence $\equiv$ in order to rearrange the order of parallel composed processes and to abstract away from the terminated process $\mathbf{0}$. We define $\equiv$ as the least congruence relation satisfying the usual axioms $P \mid Q \equiv Q \mid P$, $P \mid (Q \mid R) \equiv (P \mid Q) \mid R$, and $P \mid \mathbf{0} \equiv P$. Let

$$\mathcal{R} \quad = \quad \{(P, \sigma),\ (a.P, \sigma),\ (P \mid \prod_{1..n} b, \sigma)\}$$

We define $\mathcal{R}^+$ as the smallest relation containing $\mathcal{R}$ that is closed under process equivalence that is: if $(P, \sigma) \in \mathcal{R}^+$ and $P \equiv Q$ then $(Q, \sigma) \in \mathcal{R}^+$. We demonstrate that $\mathcal{R}^+$ is an underestimation relation. At first we note that σ has two ready sets $\{a\}$, $\{b\}$ and that $\sigma \stackrel{a}{\longmapsto} \sigma$, $\sigma \stackrel{b}{\longmapsto} \sigma$. Then we verify the pairs.

- In case of (P, σ) we have $P \searrow \{a, b\}$ thus satisfying item 1 of Definition 5.14. As regards transitions, either $P \stackrel{b}{\Longrightarrow} a.P$, $\sigma \stackrel{b}{\longmapsto} \sigma$ with $(a.P, \sigma) \in \mathcal{R}^+$, or $P \stackrel{a}{\Longrightarrow} P \mid b$, $\sigma \stackrel{a}{\longmapsto} \sigma$ with $(P \mid b, \sigma) \in \mathcal{R}^+$. Then also item 2 of Definition 5.14 is verified. We note that the same holds for pairs (Q, σ) with $Q \equiv P$. Indeed $\equiv$ preserve process transitions and ready sets.

- In case of $(a.P, \sigma)$ we have $P \searrow \{a\}$. This satisfies item 1 of Definition 5.14. Moreover, $a.P \stackrel{a}{\Longrightarrow} P$, $\sigma \stackrel{a}{\longmapsto} \sigma$ and $(P, \sigma) \in \mathcal{R}^+$. This satisfies item 2 of Definition 5.14. Similarly for (Q, σ) with $Q \equiv a.P$).

- In case of $(P \mid \prod_{1..n} b, \sigma)$ we have $P \mid \prod_{1..n} b \searrow \{a, b\}$ thus satisfying item 1 of Definition 5.14. As regard as item 2 of Definition 5.14 we have two cases. If $P \mid \prod_{1..n} b \stackrel{b}{\Longrightarrow} P'$ then $\sigma \stackrel{b}{\longmapsto} \sigma$ and it is easy to see that $(P', \sigma) \in \mathcal{R}^+$. If $P \mid \prod_{1..n} b \stackrel{a}{\Longrightarrow} P \mid \prod_{1..n+1} b$ then $\sigma \stackrel{b}{\longmapsto} \sigma$ and and $(P \mid \prod_{1..n+1} b, \sigma) \in \mathcal{R}$. Thus we conclude. Similarly for (Q, σ) with $Q \equiv P \mid \prod_{1..n} b$). $\square$

5.4.3 Overestimation

Overestimation is used for giving proper contracts to protocols implemented by clients. Then a contract overestimting describes a protocol requiring more in terms of server interaction capabilities. As for the underestimation, in this section we first define the overestimation in a set-theoretic way and then we give an operational definition. Finally we show the correspondence between the two definitions.

Definition 5.15 (Set-theoretic Overestimation) *We say that* σ overestimates P *if and only if* $\{Q : \texttt{rt}(\sigma) \between_p Q\} \subseteq \{Q : P \between_p Q\}$.

Definition 5.16 (Operational Overestimation) *A relation* $\mathcal{O}$ *is an overestimation relation if* $P\mathcal{O}\sigma$ *implies:*

1. *if* $P \searrow \mathrm{R} \neq \emptyset$ *then* $\sigma \searrow \mathrm{R}' \neq \emptyset$ *with* $\mathrm{R}' \subseteq \mathrm{R}$

2. *if* $P \stackrel{\alpha}{\Longrightarrow} P'$ *then* $\sigma \stackrel{\alpha}{\longmapsto} \sigma'$ *with* $P'\mathcal{O}\sigma'$.

Let $\dashv_o$ *be the largest overestimation relation.*

A contract σ overestimating a client protocol describes the process in more way that it is more difficult for servers to be compliant. Item 1 says that for any ready set of P – corresponding to open choices for the server – there is smaller choice on the contract. We note that, for empty ready sets of P, we do not need to find the empty ready set also in σ. Indeed, if the process decides to close the protocol, every contract is an overestimation. Item 2 deals with transitions. If $P \stackrel{\alpha}{\Longrightarrow} P'$ we require $\sigma \stackrel{\alpha}{\longmapsto} \sigma'$ with σ' that overestimates P' (for every P'). For instance, we may verify that $a+b \dashv_o a \oplus b$, and $a+b \dashv_o a.c+b.d$. On the other hand, we can also check that $a \mid b \not\dashv_u a+b+c$ (because of the item 1) $a \mid b \not\dashv_u a$ (because of the item 2). Indeed, the process $\overline{c}$ is compliant with $a+b+c$ but not with $a \mid b$. Similarly $\overline{a}+\overline{b}.c$ is compliant with a but not with $a \mid b$.

The following propositions hold.

Proposition 5.15 1. *for every* σ, $\mathsf{rec}\ x.\tau.x \dashv_o \sigma$ *and* $\mathbf{0} \dashv_o \sigma$

2. $P \dashv_o \mathbf{\Omega}$ *implies* $P \stackrel{\alpha}{\not\Longrightarrow}$

3. $P \dashv_o \mathbf{0}$ *implies* $P \stackrel{\alpha}{\not\Longrightarrow}$

4. $\alpha.P \dashv_o \alpha$ *implies* $P \stackrel{\alpha}{\not\Longrightarrow}$

Proposition 5.16 *If* $P \dashv_o \sigma$ *and* $P \stackrel{\tau}{\longrightarrow} P'$ *then* $P' \dashv_o \sigma$

Proof: By Proposition 5.9 $P \stackrel{\tau}{\longrightarrow} P' \searrow \mathrm{R}$ implies $P \searrow \mathrm{R}$. Moreover, if $P' \stackrel{\alpha}{\Longrightarrow} P''$ then $P \stackrel{\alpha}{\Longrightarrow} P''$. Therefore we conclude by the hypothesis $P \dashv_o \sigma$. □

Lemma 5.6 *If* $P \dashv_o \sigma$ *then* σ overestimates P

Proof: We have to show that if $P \dashv_o \sigma$ then $\mathtt{rt}(\sigma) \between_p Q$ implies $P \between_p Q$ (that is $\{Q : \mathtt{rt}(\sigma) \between_p Q\} \subseteq \{Q : P \between_p Q\}$). This is equivalent to demonstrate that

$$\mathcal{R} \quad = \quad \{(P,Q) : P \dashv_o \sigma,\ \mathtt{rt}(\sigma) \between_p Q\}$$

is compliance relation. By the hypothesis $P \dashv_o \sigma$ we have

$$P \searrow \mathrm{R} \neq \emptyset \text{ implies } \sigma \searrow \mathrm{R}' \neq \emptyset \text{ with } \mathrm{R}' \subseteq \mathrm{R} \tag{5.5}$$

$$P \stackrel{\alpha}{\Longrightarrow} P' \text{ implies } \sigma \stackrel{\alpha}{\longmapsto} \sigma' \text{ with } P' \dashv_o \sigma' \tag{5.6}$$

We have two cases: (a) $P \stackrel{\mu}{\not\longrightarrow}$ and (b) $P \stackrel{\mu}{\longrightarrow}$ Case (a) is immediate. In case (b) we must show (b1) $P \stackrel{\alpha}{\longrightarrow}$ and $P \stackrel{\tau}{\not\longrightarrow}$ implies $Q \Downarrow$, and (b2) $Q \parallel P \longrightarrow Q' \parallel P'$ implies $(Q', P') \in \mathcal{R}$.

As regards (b1) let assume $Q \not\Downarrow$. By $\mathtt{rt}(\sigma) \between_p Q$ we obtain $\mathtt{rt}(\sigma) \stackrel{\alpha}{\not\Longrightarrow}$ and, by (5.6), $P \stackrel{\alpha}{\not\Longrightarrow}$. This contradicts the hypothesis $P \stackrel{\alpha}{\longrightarrow}$. Hence $Q \Downarrow$.

As regards (b2) we distinguish several subcases:

$Q \stackrel{\tau}{\longrightarrow} Q'$ By the hypothesis $\mathtt{rt}(\sigma) \between_p Q$ we have $\mathtt{rt}(\sigma) \between_p Q'$. Thus and we conclude $(P, Q') \in \mathcal{R}$.

$P \stackrel{\tau}{\longrightarrow} P'$ By Proposition 5.16 we have $P' \dashv_o \sigma$. Thus we conclude $(P', Q) \in \mathcal{R}$.

$P \stackrel{\alpha}{\longrightarrow} P'$**,** $Q \stackrel{\overline{\alpha}}{\longrightarrow} Q'$ By (5.6), we have $\sigma \stackrel{\alpha}{\longmapsto} \sigma(\alpha)$ with $P' \dashv_o \sigma(\alpha)$. By Proposition 5.12, $\mathtt{rt}(\sigma) \stackrel{\alpha}{\Longrightarrow} \mathtt{rt}(\sigma(\alpha))$, and, by Proposition 5.10, $Q' \between_p \mathtt{rt}(\sigma(\alpha))$. Thus we conclude $(Q', P') \in \mathcal{R}$. $\square$

The following theorem relates contract and process compliance. It states that if the contracts approximating the client and of the server processes are compliant then provided process also comply.

Theorem 5.1 *If* $P \dashv_o \sigma_1$, $\sigma_1 \between_c \sigma_2$, *and* $Q \vdash_u \sigma_2$ *then* $P \between_p Q$.

Proof: Let $A = \{Q' : \mathtt{rt}(\sigma_1) \between_p Q'\}$ and $B = \{P' : P' \between_p \mathtt{rt}(\sigma_2)\}$. By $P \dashv_o \sigma_1$ and Lemma 5.6 we have

$$A \subseteq \{Q' : P \between_p Q'\} \tag{5.7}$$

By Proposition 5.4 $\sigma_1 \between_c \sigma_2$ implies $\mathtt{rt}(\sigma_1) \between_c \mathtt{rt}(\sigma_2)$. Thus $\mathtt{rt}(\sigma_2) \in A$ and, by (5.7), $P \between_p \mathtt{tr}(\sigma_2)$ and $P \in B$. By $\sigma_2 \vdash_u Q$ and Lemma 5.5

$$B \subseteq \{P' : P' \between_p Q\} \tag{5.8}$$

By (5.8), $P \in B$ implies $P \in \{P' : P' \between_p Q\}$. Hence we conclude $P \between_p Q$. $\square$

5.5 Regular Processes

Definition 5.17 Regular processes *are* – CCS – *processes without restriction and parallel composition.*

$$P \;::=\; \mathbf{0} \quad | \quad \mu.P \quad | \quad P + P \quad | \quad \mathsf{rec}\ x.\, P \quad | \quad x$$

We now show how to extrapolate a contract out of a regular process. The function we use is the one presented in [NH87] and we demonstrate that it is an overestimation and an underestimation of the process's protocol. Hence, it can be used for providing both server and client contracts. It mainly leaves the language unchanged but it removes internal τ actions introducing the proper internal choices. In particular we obtain internal choices whenever we have a choice with τ actions. If the choice is among processes guarded by prefixes α, only the external choice is used.

Definition 5.18 $\mathtt{tr}$ *is a function from regular processes to contracts defined by structural induction as follows:*

$$\begin{aligned}
\mathtt{tr}(\mathbf{0}) &= \mathbf{0} \\
\mathtt{tr}(\tau.P) &= \mathtt{tr}(P) \\
\mathtt{tr}(\alpha.P) &= \alpha.\mathtt{tr}(P) \\
\mathtt{tr}(P+Q) &= \begin{cases} \mathtt{tr}(P) + \mathtt{tr}(Q) & \text{if } P \stackrel{\tau}{\not\longrightarrow} \text{and} Q \stackrel{\tau}{\not\longrightarrow} \\ (\mathtt{tr}(P) + \mathtt{tr}(Q)) \oplus \bigoplus_{\substack{P \stackrel{\tau}{\longrightarrow} P' \\ Q \stackrel{\tau}{\longrightarrow} P'}} \mathtt{tr}(P') & \text{otherwise} \end{cases} \\
\mathtt{tr}(\mathsf{rec}\ x.\, P) &= \mathsf{rec}\ x.\, \mathtt{tr}(P) \\
\mathtt{tr}(x) &= x
\end{aligned}$$

For instance we may verify $\mathtt{tr}(\tau.a + \tau.b) = (a+b) \oplus (a \oplus b)$, $\mathtt{tr}(\tau.a + b) = (a+b) \oplus a$, and $\mathtt{tr}(\mathsf{rec}\ x.\, \tau.a.x + \mathsf{rec}\ y.\, b.y) = (\mathsf{rec}\ x.\, a.x + \mathsf{rec}\ y.\, b.y) \oplus \mathsf{rec}\ x.\, a.x$. As regard as divergent processes, $\mathtt{tr}(\mathsf{rec}\ x.\, \tau.x) = \mathbf{\Omega}$, and $\mathtt{tr}(\mathsf{rec}\ x.\, \tau.x + a) = \mathsf{rec}\ x.\, x + a$.

Proposition 5.17 *If $P \Downarrow$ then $\mathtt{tr}(P) \Downarrow$.*

Proof: The proof is by induction on the derivations of $P \Downarrow$. The base cases are for $P = \mathbf{0}$ and $P = \alpha.P'$. By definition of $\mathtt{tr}(\cdot)$, $\mathtt{tr}(\mathbf{0}) = \mathbf{0}$ and $\mathtt{tr}(\alpha.P') = \alpha.\mathtt{tr}(P')$. Hence we conclude by $\mathbf{0} \Downarrow$ and $\alpha.\mathtt{tr}(P') \Downarrow$. The inductive cases are:

$P = \tau.P'$ By definition of $\mathtt{tr}(\cdot)$, $\mathtt{tr}(\tau.P') = \mathtt{tr}(P')$ and, by the inductive hypothesis, $\mathtt{tr}(P') \Downarrow$. Hence we conclude.

$P = P' + P''$ We distinguish two cases. If $P' \stackrel{\tau}{\not\longrightarrow}$ and $P'' \stackrel{\tau}{\not\longrightarrow}$ then $\mathtt{tr}(P'+P'') = \mathtt{tr}(P')+ \mathtt{tr}(P'')$ and we conclude by the inductive hypothesis. Otherwise $\mathtt{tr}(P' + P'') = (\mathtt{tr}(P')+\mathtt{tr}(P''))+\oplus\bigoplus_{P' \stackrel{\tau}{\longrightarrow} Q, P'' \stackrel{\tau}{\longrightarrow} Q} \mathtt{tr}(Q)$. By the hypothesis $P'+P'' \Downarrow$ it is easy to see that $P' \Downarrow$ and $P'' \Downarrow$. This implies $Q \Downarrow$ for every Q. Thus we conclude by the inductive hypothesis.

$P = \mathsf{rec}\ x.\,P'$ By the hypothesis $P \Downarrow$, we have $P'\{^P/_x\} \Downarrow$ and, by the inductive hypothesis, $\mathtt{tr}(P'\{^P/_x\}) \Downarrow$. Since $\mathtt{tr}(P) = \mathsf{rec}\ x.\,\mathtt{tr}(P')$, we reduce to demonstrate $\mathtt{tr}(P')\{^{\mathtt{tr}(P)}/_x\} \Downarrow$. We conclude by $\mathtt{tr}(P')\{^{\mathtt{tr}(P)}/_x\} = \mathtt{tr}(P'\{^P/_x\})$ and by the inductive hypothesis. $\square$

Proposition 5.18 *If $P \searrow \mathrm{R}$ then $\mathtt{tr}(P) \searrow \mathrm{R}$.*

Proof: The proof is by induction on the derivations of $P \searrow \mathrm{R}$. In case of divergence we conclude by Proposition 5.17. The base cases are for $P = \mathbf{0}$ and $P = \alpha.P'$. Since $\mathtt{tr}(\mathbf{0}) = \mathbf{0}$ and $\mathtt{tr}(\alpha.P') = \alpha.\mathtt{tr}(P')$ we conclude by definition of ready set. The inductive cases are:

$P = \tau.P'$ By the definition of ready set of processes $\tau.P' \searrow \mathrm{R}$ implies $P' \searrow \mathrm{R}$. Since $\mathtt{tr}(\tau.P') = \mathtt{tr}(P')$, by the inductive hypothesis we conclude $\mathtt{tr}(P') \searrow \mathrm{R}$.

$P = P' + P''$ We distinguish two cases. If $P' \stackrel{\tau}{\not\longrightarrow}$ and $P'' \stackrel{\tau}{\not\longrightarrow}$ then $\mathtt{tr}(P'+P'') = \mathtt{tr}(P')+ \mathtt{tr}(P'')$ and we conclude by the inductive hypothesis. Otherwise $\mathtt{tr}(P' + P'') = (\mathtt{tr}(P') + \mathtt{tr}(P'')) \oplus \bigoplus_{P' \stackrel{\tau}{\longrightarrow} Q, P'' \stackrel{\tau}{\longrightarrow} Q} \mathtt{tr}(Q)$. By definition of ready set of processes $P' + P'' \searrow \mathrm{R}$ implies either (1) $P' \stackrel{\tau}{\longrightarrow} Q' \searrow \mathrm{R}$ or (2) $P'' \stackrel{\tau}{\longrightarrow} Q' \searrow \mathrm{R}$ or (3) $P' \searrow \mathrm{R}'$, $Q' \searrow \mathrm{R}''$ with $\mathrm{R} = \mathrm{R}' \cup \mathrm{R}''$. In all the cases we conclude by the inductive hypothesis and by definition of ready set of contracts.

$P = \mathsf{rec}\ x.\, P'$ then by the hypothesis $P \searrow \mathrm{R}$ we have $P'\{^P/_x\} \searrow \mathrm{R}$. Since $\mathtt{tr}(P) = \mathsf{rec}\ x.\, \mathtt{tr}(P')$, we reduce to demonstrate $\mathtt{tr}(P')\{^{\mathtt{tr}(P)}/_x\} \searrow \mathrm{R}$. We conclude by $\mathtt{tr}(P')\{^{\mathtt{tr}(P)}/_x\} = \mathtt{tr}(P'\{^P/_x\})$ and by the inductive hypothesis. $\square$

The following Proposition is not immediate by the definition of $\overset{\alpha}{\longmapsto}$ because the transition relation is defined by induction on the structure.

Proposition 5.19 *Let $\sigma = \mathsf{rec}\ x.\, \sigma_1$. If $\sigma_1\{^\sigma/_x\} \overset{\alpha}{\longmapsto} \sigma_2$ then $\sigma \overset{\alpha}{\longmapsto} \sigma_2$.*

Proof: The Proposition follows by induction on the structure of σ_1. $\square$

Proposition 5.20 *If $P \overset{\alpha}{\Longrightarrow} P'$ then $\mathtt{tr}(P) \overset{\alpha}{\longmapsto} \bigoplus_{i\in I} \mathtt{tr}(Q_i)$ with $Q_j = P'$ for some $j \in I$.*

Proof: The proof is by induction on the derivations of $P \overset{\alpha}{\Longrightarrow} P'$. The base case is $P = \alpha.P'$. Then $\mathtt{tr}(\alpha.P') = \alpha.\mathtt{tr}(P')$ and we conclude. The inductive case are:

$P = \tau.Q$ By $P \overset{\alpha}{\Longrightarrow} P'$ we have $Q \overset{\alpha}{\Longrightarrow} P'$. Since $\mathtt{tr}(P) = \mathtt{tr}(Q)$, we conclude by the inductive hypothesis.

$P = Q' + Q''$ We distinguish two cases: (a) $Q' \overset{\tau}{\not\longrightarrow}$, $Q'' \overset{\tau}{\not\longrightarrow}$ and (b) either $Q' \overset{\tau}{\longrightarrow}$ or $Q'' \overset{\tau}{\longrightarrow}$.

In case (a), $\mathtt{tr}(Q' + Q'') = \mathtt{tr}(Q') + \mathtt{tr}(Q'')$. By $Q' + Q'' \overset{\alpha}{\longrightarrow} P'$ we obtain either: (a1) $Q' \overset{\alpha}{\longrightarrow} P'$ or (a2) $Q'' \overset{\alpha}{\longrightarrow} P'$. We discuss (a1), (a2) is similar. If $\mathtt{tr}(Q'') \overset{\alpha}{\not\longmapsto}$ then $\mathtt{tr}(Q' + Q'') \overset{\alpha}{\longmapsto} \bigoplus_{i\in I} \mathtt{tr}(R_i)$ and we conclude. If $\mathtt{tr}(Q'') \overset{\alpha}{\longmapsto} \sigma'$ then $\mathtt{tr}(Q' + Q'') \overset{\alpha}{\longmapsto} \bigoplus_{i\in I} \mathtt{tr}(R_i) \oplus \sigma'$ and we conclude.

In case (b) $\mathtt{tr}(Q'+Q'') = (\mathtt{tr}(Q')+\mathtt{tr}(Q''))\oplus\bigoplus_{\substack{Q' \overset{\tau}{\longrightarrow} R \\ Q'' \overset{\tau}{\longrightarrow} R}} \mathtt{tr}(R)$. We distinguish four subcases: (b1) $Q' \overset{\alpha}{\longrightarrow} P'$, (b2) $Q' \overset{\tau}{\longrightarrow}\overset{\alpha}{\Longrightarrow} P'$, (b3) $Q'' \overset{\alpha}{\longrightarrow} P'$, (b4) $Q'' \overset{\tau}{\longrightarrow}\overset{\alpha}{\longrightarrow} P'$. Case (b1) and (b3) are similar to (a1) and (a2). We discuss (b2), (b4) is similar. If $\mathtt{tr}(Q'') \overset{\alpha}{\not\longmapsto}$ then, by the inductive hypothesis, $\mathtt{tr}(Q') \overset{\alpha}{\longmapsto} \bigoplus_{i\in I} \mathtt{tr}(R_i)$ with $R_j = P'$. Then $\mathtt{tr}(Q' + Q'') \overset{\alpha}{\longmapsto} \bigoplus_{i\in I} \mathtt{tr}(R_i) \oplus \bigoplus_{i\in I} \mathtt{tr}(R_i)$ and we conclude. If $\mathtt{tr}(Q'') \overset{\alpha}{\not\longmapsto} \sigma$ then $\mathtt{tr}(Q' + Q'') \overset{\alpha}{\longmapsto} \bigoplus_{i\in I} \mathtt{tr}(R_i) \oplus \sigma$ and we conclude.

$P = \mathsf{rec}\ x.\,Q$ By the inductive hypothesis, $\mathtt{tr}(Q\{^P/_x\}) \stackrel{\alpha}{\longmapsto} \bigoplus_{i\in I} \mathtt{tr}(Q_i)$ with $Q_j = P'$ for some $j \in I$. By $\mathtt{tr}(Q\{^P/_x\}) = \mathtt{tr}(Q)\{^{\mathtt{tr}(P)}/_x\}$ and by Proposition 5.19, we have $\mathtt{tr}(P) = \mathsf{rec}\ x.\,\mathtt{tr}(Q) \stackrel{\alpha}{\longmapsto} \bigoplus_{i\in I} \mathtt{tr}(Q')$. Thus we conclude. $\square$

Lemma 5.7 (Soundness) $P \vdash_u \mathtt{tr}(P)$ *and* $\mathtt{tr}(P) \dashv_o P$

Proof: As regard as $P \vdash_u \mathtt{tr}(R)$ let

$$\mathcal{R} = \{(P, \mathtt{tr}(P)) : P \text{ is a regular process}\}$$

and let $\mathcal{R}^+$ be the least relation containing $\mathcal{R}$ and closed under the following operation:

- if $(P, \mathtt{tr}(P)) \in \mathcal{R}^+$ then $(P, \mathtt{tr}(P) \oplus \sigma') \in \mathcal{R}^+$

We show that $\mathcal{R}^+$ is an underestimation relation and an overestimation relation. We distinguish two cases:

$(P, \mathtt{tr}(P))$ By Proposition 5.18, condition 2.a of Definition 5.14 and condition 1 of Definition 5.16 are verified. By Proposition 5.20 if $P \stackrel{\alpha}{\Longrightarrow} P'$ then either $\mathtt{tr}(P) \stackrel{\alpha}{\longmapsto} \mathtt{tr}(P')$ or $\mathtt{tr}(P) \stackrel{\alpha}{\longmapsto} \mathtt{tr}(P') \oplus \sigma$. We conclude by $(P', \mathtt{tr}(P'))$ and $(P', \mathtt{tr}(P') \oplus \sigma)$.

$(P, \mathtt{tr}(P) \oplus \sigma)$ By Proposition 5.18, condition 2.a of Definition 5.14 and condition 1 of Definition 5.16 are verified. By Proposition 5.20, if $P \stackrel{\alpha}{\Longrightarrow} P'$ then either $\mathtt{tr}(P) \oplus \sigma \stackrel{\alpha}{\longmapsto} \mathtt{tr}(P')$ or $\mathtt{tr}(P) \oplus \sigma \stackrel{\alpha}{\longmapsto} \mathtt{tr}(P') \oplus \sigma$. We conclude by $(P', \mathtt{tr}(P'))$ and $(P', \mathtt{tr}(P') \oplus \sigma)$. $\square$

Part IV

Conclusions

Chapter 6

Conclusions

In this thesis we have started an investigation aimed at the formal definition of a data contract language for describing the data format of messages exchanged between Web services and a simple behavioral contract language suitable for describing interactions of clients with Web services.

Regarding the description of Web services interfaces with the schema language, it is remarkable that WSDL 1.1 (already published as a W3C Note) does not consider service references as first class values, that is natural in a distributed setting. This lack of expressivity has been at least partly amended in WSDL 2.0 that, at the time of this writing, is in a Proposed Recommendation status. Still, we note significant differences between our approach and the way "Web services as values" are handled in WSDL 2.0. For example the client receiving a service reference must eventually compare the schema of its WSDL with some local schema before using it or forward it to a third party. While this comparison, called subschema relation in our schema language, is fundamental to obtain the needed flexibility, it has been overlooked in WSDL 2.0. Another difference is in the meaning of service references. In WSDL 2.0, on the other hand, it is the whole Web service, not a specific operation, that is communicated, and it is up to the receiver to invoke the desired operation(s). In our schema language we have a finer granularity that allow us to communicate also references to remote operations.

Few remarks about XML-SCHEMA are in order. First of all there is a large overlapping between XML-SCHEMA and the our schema language, which has been formalized in Chapter 3 Section 3.3. Apart from references, the other major departure from XML schema is

the support for nondeterministic labeled schemas. These schemas make the computational complexity of the subschema relation exponential, but they are important for the static semantics of programming language. For instance in order to verify the exhaustiveness of the pattern-matching statement in XDuce, CDuce, and PiDuce non-deterministic schemas are used. Noticeably, the constraint of labelled-determinedness on channel schemas guarantees a polynomial cost for the subschema relation (and for the pattern matching) at runtime.

The schema language is not adequate for describing conversation protocols. Thus, we presented a formal contract language suitable for describing interactions of clients with Web services. We have defined a precise notion of compatibility between services, called subcontract relation, so that equivalent services can be safely replaced with each other. This notion of compatibility is immediately applicable in any query-based system for service discovery, as well as for verifying that a service implementation respects its interface. To the best of our knowledge, this relation is original and it does not coincide with either must, or may, or testing preorders. Based on the subcontract relation, we have provided a formal notion of compliance, such that clients that are verified to be compliant with a contract are guaranteed to successfully complete the interaction with any service that exports that contract.

We leave to future works the design of more concrete algorithms for querying. A straightforward implementation of our compliance relation in repositories will inspect any service contract verifying the subcontract relation with the contract of the client. This will be quite slow and several optimizations are possible. For instance the repository can preorder contracts returning more than one contract with a single check. Since our subcontract relation is not transitive it is not suitable for such an ordering. Probably the best candidate for this purpose is the transitive closure of $\preceq$ [CGP07].

In this thesis, normal forms and dual contracts are simple functions applied to the contracts. Their definition rely on the definitions of ready sets and on the transition relation. Alternatively, it is possible to define the normed and the dual contracts through a set of axioms. In particular, in [Hen88] the normal form of finite contracts is obtained through the axioms in Chapter 4 Section 4.3.3. It would be interesting to extend Table 4.2 with axioms for dealing with recursive contracts and then verifying that the new axioms allow the tranformation of a contract to its normal form.

A limitation of our contract language is that channels cannot be communicated. The passage from CCS-like formalism to a π-calculus formalism looks crucial for more than one reason. First it will allow us to take into account and generalize the forthcoming versions of WSDL. Also, it will more faithfully mimic WSCL protocols which discriminate on the content of messages. Besides, the type of these parameters could also be used to define contract isomorphisms to improve service discovery. In particular we will study *provable* isomorphisms, that is, isomorphisms for which it is possible to exhibit a process that "converts" the two contracts: for instance, imagine that we search for a service that implements the contract `In(Int).In(Int)`, that is, a service that sequentially waits twice for an integer on the port `In`; the query may return a reference to a service with a contract isomorphic to it, say, `In(Int×Int)` together with a process that "proves" that these two contracts are isomorphic, that is, in the specific case, a process that buffers the two inputs and sends the pair of them on `In`: by composing this process with the original client (written for the first contract) one obtains a client complying with the discovered service.

On the linguistic side we would like to explore new process constructions that could take into account information available with contracts. For instance imagine a client that wants to use a service exporting the contract $(a + b) \oplus a$; in the process language, client cannot specify that it wants to connect with b if available, and on a otherwise. We want also to devise query languages for service discovery, in particular we aim to devise a simple set-theoretic interpretation of contracts as sets of processes, use it to add union, intersection, and negation operators for contracts, and subsequently use these as query primitives.

A final issue brought by higher-order and whose exploration looks promising is that higher-order channels will allow us to use a continuation passing style (CPS) of programming. It is well-known that CPS can be used for stateless implementation of interactive web-sessions [Que03], thus we plan to transpose such a technique to contracts and resort to CPS to describe stateful interactions of services.

References

[AB05] Lucia Acciai and Michele Boreale. XPi: a typed process calculus for XML messaging. In *7th Formal Methods for Object-Based Distributed Systems (FMOODS'05)*, volume 3535 of *Lecture Notes in Computer Science*, pages 47 – 66. Springer-Verlag, 2005.

[AC93] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, September 1993.

[ACE+04] Philippe Altherr, Vincent Cremet, Burak Emir, Stphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, Matthias Zenger, and Martin Odersky. The scala programming language: Documentation. Available at `http://scala.epfl.ch/docu/index.html`, 2004.

[ACJ04] Frank Atanassow, Dave Clarke, and Johan Jeuring. UUXML: A type-preserving XML Schema–Haskell data binding. In Bharat Jayaraman, editor, *Practical Aspects of Declarative Languages, 6th International Symposium, PADL 2004, Dallas, TX, USA, June 2004, Proceedings*, number 3057 in LNCS, pages 71–85, Berlin Heidelberg, June 2004. Springer–Verlag.

[BBB+02] Arindam Banerji, Claudio Bartolini, Dorothea Beringer, Venkatesh Chopella, et al. *Web Services Conversation Language (WSCL) 1.0*, mar 2002.

[BBM+01] Keith Ballinger, Peter Brittenham, Ashok Malhotra, William A. Nagy, and Stefan Pharies. Web Services Inspection Language: Specification. Available

at `http://www-128.ibm.com/developerworks/library/specification/ws-wsilspec/`, November 2001.

[BCF03] Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. CDuce: an XML-centric general-purpose language. In *Proceedings of the 8th ACM SIGPLAN International Conference on Functional Programming (ICFP-03)*, pages 51–63, New York, 2003. ACM Press.

[BG06] Michele Boreale and Fabio Gadducci. Processes as formal power series: A coinductive approach to denotational semanticss. *Theoretical Computer Science*, 360:440–458, 2006.

[BGZ03] Nadia Busi, Maurizio Gabbrielli, and Gianluigi Zavattaro. Replication vs. recursive definitions in channel based calculi. In *Colloquium on Automata, Languages and Programming (ICALP)*, pages 133–144, 2003.

[BGZ04] Nadia Busi, Maurizio Gabbrielli, and Gianluigi Zavattaro. Comparing recursion, replication, and iteration in process calculi. In *Colloquium on Automata, Languages and Programming (ICALP)*, pages 307–319, 2004.

[BH97] Micheal Brandt and Friz Henglein. Coinductive axiomatization of recursive type equality and subtyping. In Roger Hindley, editor, *3rd International Conference on Typed Lambda Calculi and Application (TLCA), Nancy, France*, volume 1210 of *Lecture Notes in Computer Science*, pages 63 – 81. Springer-Verlag, April 1997. Full version in Fundamenta Informaticae, Vol. 33, pp. 309-338, 1998.

[BKL01] D. Beringer, H. Kuno, and M. Lemon. *Using* WSCL *in a* UDDI *Registry 1.0*, 2001. UDDI Working Draft Best Practices Document.

[BL06] David Booth and Canyang Kevin Liu. *Web Services Description Language (WSDL) Version 2.0 Part 0: Primer*, March 2006.

[BLM05] A. Brown, C. Laneve, and L.G. Meredith. `PiDuce`: A process calculus with native XML datatypes. In *Proceedings of 2nd International Workshop on Web Services and Formal Methods*, LNCS. Springer-Verlag, 2005.

[BP00] Peter Buneman and Benjamin Pierce. Union types for semistructured data. *Lecture Notes in Computer Science*, 1949:184–??, 2000.

[Cas06] Giuseppe Castagna. Private communications with giuseppe castagna, 2006.

[CCLP06] Samuele Carpineti, Giuseppe Castagna, Cosimo Laneve, and Luca Padovani. A formal account of contracts for web services. In *3-rd Int. Workshop on Web Services and Formal Methods (WS-FM 2006)*, LNCS. Springer-Verlag, 2006.

[CDG+97] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available at `http://www.grappa.univ-lille3.fr/tata`, 1997. released October, 1st 2002.

[CGK+02] F. Curbera, Y. Goland, J. Klein, F. Leyman, D. Roller, S. Thatte, and S. Weerawarana. Business process execution language for web services (bpel4ws 1.0). 2002.

[CGP07] Giuseppe Castagna, Nils Gesbert, and Luca Padovani. A Theory of Contracts for Web Services. In *ACM SIGPLAN Workshop on Programming Language Technologies for XML (PLAN-X 2007)*, 2007. To appear.

[Cha03] Don Chamberlin. XQuery: a query language for XML. In ACM, editor, *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data 2003*, pages 682–682. ACM Press, 2003.

[CHL+06] Roberto Chinnici, Hugo Haas, Amelia A. Lewis, Jean-Jacques Moreau, et al. *Web Services Description Language (WSDL) Version 2.0 Part 2: Adjuncts*, mar 2006.

[CHYa] Marco Carbone, Kohei Honda, and Nabuko Yoshida. Structured Global Programming for Communication Behaviour. Available at `http://www.dcs.qmul.ac.uk/~carbonem/cdlpaper/SGPCBmiddle.pdf`.

[CHY+b] Marco Carbone, Kohei Honda, Nabuko Yoshida, Robin Milner, Gary Brown, and Steve Ross Talbot. A Theoretical Basis of Communication-Centred Concurrent Programming. Available as W3C note at `http://www.dcs.qmul.ac.uk/~carbonem/cdlpaper/workingnote.pdf`.

[CJ04] J. Colgrave and K. Januszewski. Using wsdl in a uddi registry, version 2.0.2. Technical note, OASIS, 2004.

[CL06] Samuele Carpineti and Cosimo Laneve. A basic contract language for Web services. In *Proceedings of the European Symposium on Programming (ESOP 2006)*, LNCS. Springer-Verlag, 2006.

[CLM05] Samuele Carpineti, Cosimo Laneve, and Paolo Milazzo. `BoPi` – a project for experimenting web services technologies. In Jorg Desel and Yosinori Watanabe, editors, *5th Application of Concurrency to System Design, (ACSD 2005)*. IEEE Computer Society Press, 2005.

[CLP06] Samuele Carpineti, Cosimo Laneve, and Luca Padovani. PiDuce – a project for experimenting web services technologies. Available at `http://www.cs.unibo.it/PiDuce/`, 2006.

[CM01] J. Clark and M. Murata. RELAX NG Specification. Available at `http://www.oasis-open.org/committees/relax-ng/spec-20011203.html`, 2001. December, 3rd 2001.

[CMRW06] Roberto Chinnici, Jean-Jacques Moreau, Arthur Ryman, and Sanjiva Weerawarana. *Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language*, mar 2006. Available at `http://www.w3.org/TR/2006/CR-wsdl20-20060327`.

[CNV05] G. Castagna, R. De Nicola, and D. Varacca. Semantic subtyping for the π-calculus. In *20th IEEE Symposium on Logic in Computer Science (LICS'05)*. IEEE Computer Society, 2005.

[Com04] Python Community. Python/xml documentation. Available at `http://pyxml.sourceforge.net/`, 2004.

[Cos95] R. Di Cosmo. *Isomorphisms of Types: from Lambda Calculus to Information Retrieval and Language Desig.* Birkhauser, 1995. ISBN-0-8176-3763-X.

[Cov02] Cover Pages. Web services inspection language (wsil): Technology report. Available at `http://xml.coverpages.org/wsil.html`, October 2002.

[FHRR04] Cédric Fournet, C. A. R. Hoare, Sriram K. Rajamani, and Jakob Rehof. Stuck-free conformance. Technical Report MSR-TR-2004-69, Microsoft Research, July 2004.

[fWIFT] Society for Worldwide Interbank Financial Telecommunication. SWIFT. Available at http://www.swift.com/.

[GH99] Simon J. Gay and Malcolm Hole. Types and subtypes for client-server interactions. In *European Symposium on Programming*, pages 74–90, 1999.

[GLPS04] Vladimir Gapeyev, Michael Levin, Benjamin C. Pierce, and Alan Schmitt. The Xtatic Project: Native XML processing for C#. Available at http://scala.epfl.ch/docu/index.html, 2004.

[GNO03] Yaron Goland, Mark Nottingham, and David Orchard. WS-CallBack Protocol (WS-CallBack). Available at http://dev2dev.bea.com/webservices/WS-CallBack-0_9.html, 2003.

[Gro] Web Services Description Working Group. Web Services Description Language (WSDL) 1.1). Available at http://www.w3.org/TR/2001/NOTE-wsdl-20010315. W3C Note 15 March 2001.

[Gro03] XML Protocol Working Group. SOAP Version 1.2 Part 1: Messaging Framework. Available at http://www.w3.org/TR/2003/REC-soap12-part1-20030624/, 2003. June, 24th 2003.

[Gro04a] OMG Object Management Group. Common Object Request Broker Architecture (CORBA/IIOP). Available at http://www.omg.org/technology/documents/formal/corba_iiop.htm, 2004.

[Gro04b] W3C XML Schema Working Group. XML Schema Part 1: Structures Second Edition. Available at http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/structures.html, 2004. W3C Recommendation - October, 28th 2004.

[Gro04c] W3C XML Schema Working Group. XML Schema Part 2: Datatypes Second Edition. Available at `http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/datatypes.html`, 2004. W3C Recommendation - October, 28th 2004.

[Gro04d] Web Service Activity Group. Web services architecture requirements. Available at `http://www.w3.org/TR/wsa-reqs/`, 2004. Febraury, 11th 2004.

[Gro04e] Web Services Addressing Working Group. Web Services Addressing (WS-Addressing). Available at `http://www.w3.org/Submission/2004/SUBM-ws-addressing-20040810/`, 2004. August, 10th 2004.

[Gro04f] XML Protocol Working Group. Extensible Markup Language (XML) 1.0 (third edition). Available at `http://www.w3.org/TR/2004/REC-xml-20040204`, 2004. February, 4th 2004.

[Hen85] M. Hennessy. Acceptance trees. *JACM: Journal of the ACM*, 32(4):896–928, 1985.

[Hen88] M. C. B. Hennessy. *Algebraic Theory of Processes*. Foundation of Computing. MIT Press, 1988.

[Hoa04] C. A. R. Hoare. *Communicating Sequential Processes*. Jim Davies, June 2004.

[HP03] Haruo Hosoya and Benjamin C. Pierce. XDuce: A statically typed XML processing language. *ACM Transactions on Internet Technology (TOIT)*, 3(2):117–148, 2003.

[HVK98] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. *Lecture Notes in Computer Science*, 1381:122–??, 1998.

[HVP00a] Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. Regular expression types for XML. *ACM SIGPLAN Notices*, 35(9):11–22, 2000.

[HVP00b] Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. Regular expression types for XML. *ACM SIGPLAN Notices*, 35(9):11–22, 2000.

[Mic05] Sun Microsytems. The Java Language Specification: Third edition. Available at http://java.sun.com/docs/books/jls/third_edition/html/j3TOC.html, 2005.

[Mic06] Microsoft Corporation. .net. Available at http://www.microsoft.com/net/, 2006.

[Mil82] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982.

[Mil91] R. Milner. The Polyadic π-Calculus: A Tutorial. Technical report, University of Edinburgh, October 1991.

[MLM01] M. Murata, D. Lee, and M. Mani. Taxonomy of XML schema languages using formal language theory. In *Extreme Markup Languages*, Montreal, Canada, 2001.

[NH84] Rocco De Nicola and Matthew Hennessy. Testing equivalences for processes. *Theor. Comput. Sci*, 34:83–133, 1984.

[NH87] Rocco De Nicola and Matthew Hennessy. CCS without tau's. In *TAPSOFT '87/CAAP '87: Proceedings of the International Joint Conference on Theory and Practice of Software Development, Volume 1: Advanced Seminar on Foundations of Innovative Software Development I and Colloquium on Trees in Algebra and Programming*, pages 138–152, London, UK, 1987. Springer-Verlag.

[Phi87] I. Phillips. Refusal testing. *Theoretical Computer Science*, 50(3):241–284, 1987.

[PS93] Benjamin C. Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. In *Logic in Computer Science*, 1993. Full version in *Mathematical Structures in Computer Science* , Vol. 6, No. 5, 1996.

[Que03] Christian Queinnec. Inverting back the inversion of control or, continuations versus page-centric programming. *SIGPLAN Not.*, 38(2):57–64, 2003.

[Rit93] Mikael Rittri. Retrieving library functions by unifying types modulo linear isomorphism. *RAIRO Theoretical Informatics and Applications*, 27(6):523–540, 1993.

[RR02] Sriram K. Rajamani and Jakob Rehof. Conformance Checking for Models of Asynchronous Message Passing Software. In *Proceedings CAV 02, International Conference on Computer Aided Verification*, 2002.

[Sav05] Savas Parastatidis and Jim Webber. *MEP SSDL Protocol Framework*, April 2005. Available at `http://ssdl.org`.

[SP] J. Webber S. Parastatidis. *The SOAP Service Description Language*. Available at `http://www.ssdl.org`.

[Spe02] OASIS UDDI Committee Specification. UDDI Version 2.04 api Specification. Available at `http://uddi.org/pubs/ProgrammersAPI-V2.04-Published-20020719.htm`, 2002. July, 19th 2002.

[Sun06] Sun Microsystem. Java2 entreprise edition. Available at `http://java.sun.com/javaee/`, 2006.

[THK94] Kaku Takeuchi, Kohei Honda, and Makoto Kubo. An interaction-based language and its typing system. In *Parallel Architectures and Languages Europe*, pages 398–413, 1994.

[Uni] United Nations Directories for Electronic Data Interchange for Administration: Commerce and Transport. UN/EDIFACT. Avaialble at `http://www.unece.org/trade/untdid/welcome.htm`.

[Vin04] Steve Vinosky. Web service notifications. *IEEE Internet Computing magazine*, 2004.