Abstract

Service-oriented computing highly supports the development of future business applications through the use of (Web) services. Two main challenges for Web services are the aggregation of services into new (complex) business applications, and the adaptation of services presenting various types of interaction mismatches.

The ultimate objective of this thesis is to define a methodology for the semi-automated aggregation and adaptation of Web services capable of suitably overcoming semantic and behaviour mismatches in view of business process integration within and across organisational boundaries.

We tackle the aggregation and adaptation of services described by service contracts, which consist of signature (WSDL), ontology information (OWL), and behaviour specification (YAWL). We first describe an aggregation technique that automatically generates contracts of composite services satisfying (behavioural) client requests from a registry of service contracts. Further on, we present a behaviour-aware adaptation technique that supports the customisation of services to fulfil client requests. The adaptation technique can be used to adapt the behaviour of services to satisfy both functional and behavioural requests.

In order to support the generation of service contracts from real-world service descriptions, we also introduce a pattern-based compositional translator for the automated generation of YAWL workflows from BPEL business processes. In this way, we pave the way for the formal analysis, aggregation, and adaptation of BPEL processes.
To my family and to Guaci.
I would like to express my gratitude towards Professor Antonio Brogi for all his support, and for the numerous fruitful discussions we had over the past few years.

Special thanks go to Professor Roberto Bruni and Professor Maria Simi for their feedback as internal reviewers of this thesis, as well as to Professor Boualem Bena-tallah and to Doctor Francisco Curbera for their feedback as external reviewers.

Last but not least, I would like to thank Sara Corfini, Andrea Lorenzani, and Matteo Tanca for the collaborations we had on discovering Web Services.
Contents

1 Introduction ......................................................... 13
  1.1 Context and Motivations .................................... 13
    1.1.1 Web Services: Main Ingredients .................... 13
    1.1.2 Web Services: Some Open Issues .................. 15
  1.2 Main Contributions of the Thesis ......................... 18
  1.3 Structure of the Thesis .................................. 20

2 Background: YAWL (Yet Another Workflow Language) ......... 23

3 Web Service Aggregation ........................................ 29
  3.1 Motivating Example ....................................... 31
  3.2 Overview of the Aggregation Phases ...................... 33
  3.3 Service Contracts ....................................... 34
  3.4 Description of the Aggregation Approach ................. 36
    3.4.1 Reachability Analysis ............................... 36
    3.4.2 Service Execution Traces ........................... 40
    3.4.3 Service Matching .................................. 42
    3.4.4 Core Aggregation and Contract Generation .......... 46
    3.4.5 Contract Validation ................................ 53
    3.4.6 Informal Proof of Correctness ...................... 57
  3.5 Examples .................................................. 57
    3.5.1 First Example ....................................... 57
    3.5.2 Second Example ..................................... 65
    3.5.3 Third Example ...................................... 70
    3.5.4 Fourth Example ..................................... 78
  3.6 Implementation ........................................... 82
    3.6.1 Implementation Choices. ............................ 82
    3.6.2 Main Implementation Solutions ...................... 84
    3.6.3 Code Structure and Code Quality Evaluation .......... 85
  3.7 Complexity Analysis ...................................... 86
  3.8 Middleware Aspects ....................................... 87
  3.9 Related Work ............................................. 90
  3.10 Discussion ................................................ 96
CHAPTER 0. CONTENTS

4 Web Service Adaptation ........................................ 99
  4.1 Functional Service Adaptation .................................. 101
    4.1.1 Motivating Example ........................................ 101
    4.1.2 Matching the Clothing Shop Service .......................... 103
    4.1.3 Contract Generation .......................................... 107
    4.1.4 Informal Proof of Correctness ............................... 109
  4.2 Behavioural Service Adaptation .................................. 110
    4.2.1 Motivating Example ........................................ 110
    4.2.2 Adaptation Phases .......................................... 112
    4.2.3 Informal Proof of Correctness ............................... 125
  4.3 Complexity Analysis ............................................ 127
  4.4 Related Work .................................................. 128
  4.5 Discussion .................................................... 130

5 Translating BPEL processes into YAWL workflows .............. 133
  5.1 A Brief Introduction to BPEL ................................... 135
  5.2 From BPEL to YAWL .............................................. 138
    5.2.1 Patterns of BPEL Basic Activities .......................... 139
    5.2.2 Patterns of BPEL Structured Activities .................... 151
    5.2.3 BPEL Processes ............................................. 174
  5.3 Example: Complete Translation of a (Simple) BPEL Process ... 178
    5.3.1 Generating the GCD Workflow ............................... 178
    5.3.2 Use Case of the GCD Workflow .............................. 187
  5.4 Related Work .................................................. 187
  5.5 Discussion .................................................... 188

6 Concluding Remarks ............................................ 191

Bibliography .................................................... 197
List of Tables

3.1 TTs of the example workflows. ........................................... 41
3.2 TT for the workflow in Figure 3.4. ........................................... 41
3.3 Data-flow mapping for \{SE, FD, FA\}. ................................. 45
3.4 Data-flow mapping for \{FS, FD, FA\}. ................................. 45
3.5 Data-flow mapping for \{FS, FA\}. ................................. 45
3.6 TT of the aggregated service AggS$_1$ (Figure 3.15). ............ 55
3.7 TT of the aggregated service AggS$_2$ (Figure 3.16). ............ 55
3.8 TT of the aggregated service AggS$_3$ (Figure 3.17). ............ 56
4.1 TT for the Clothing Shop service. ........................................... 105
List of Figures

2.1 Examples of YAWL workflows. ......................................... 28
3.1 Service contract levels. .................................................. 35
3.2 Motivating example workflows with explicit conditions. ........ 38
3.3 RGs of the four workflows in Figure 3.2. .......................... 39
3.4 Workflow with cycle. .................................................... 41
3.5 Matchmaker Graph for our example. ............................... 44
3.6 Expansion of the Get Filename task of the File Server service. .. 48
3.7 Control-flow analysis for the File Server service. ................. 49
3.8 Control-flow analysis for the Search Engine service. ............. 49
3.9 Control-flow analysis for the File Downloader service. ........... 49
3.10 Control-flow analysis for the Fetch Application service. ......... 50
3.11 Data-flow analysis for the three possible scenarios of our example. 50
3.12 AggS1: Workflow obtained by the aggregation of SE, FD, and FA. 51
3.13 AggS2: Workflow obtained by the aggregation of FS, FD, and FA. 51
3.14 AggS3: Workflow obtained by the aggregation of FS and FA. .... 52
3.15 Optimised workflow and RG for the AggS1 service. ............... 53
3.16 Optimised workflow and RG for the AggS2 service. ............... 54
3.17 Optimised workflow and RG for the AggS3 service. ............... 55
3.18 Example illustrating the workflows of three Web services. ....... 58
3.19 IO matches of the three services. ...................................... 61
3.20 Task Expansion step applied to the three example workflows. .... 62
3.21 Applying the Control-Flow step on the example workflows. ....... 64
3.22 Transforming IO matches into dependencies btw. IDs and ODs. ... 65
3.23 Rough YAWL workflow of the aggregated service. ............... 66
3.24 Final YAWL workflow of the aggregated service. .................. 67
3.25 Tackling composite tasks. ............................................ 68
3.26 Task Expansion step applied to the Bank2 service. ............... 69
3.27 Control-Flow Analysis of the example workflows in Figure 3.25. .. 69
3.28 New IO matches table for the services in Figure 3.25. ............. 70
3.29 Transforming IO matches into dependencies btw. IDs and ODs. ... 71
3.30 Rough aggregated workflow for the services in Figure 3.25. ....... 72
3.31 Final aggregated workflow for the services in Figure 3.25. ..... 73
3.32 Tackling multiple-instance tasks. ................................... 74
3.33 Expanding the GetBookPrices task of the BookStore2 workflow. . . . 74
3.34 Control-Flow Analysis of the example workflows in Figure 3.32. . . . 75
3.35 New IO matches table for the services in Figure 3.32. . . . . . . . 76
3.36 Transforming IO matches into dependencies btw. IDs and ODs. . . . 77
3.37 Rough aggregated workflow for the services in Figure 3.32. . . . . . 78
3.38 Final aggregated workflow for the services in Figure 3.32. . . . . . 79
3.39 Tackling cancellation sets. . . . . . . . . . . . . . . . . . . . . . . . . 80
3.40 Expanding tasks included in, or associated to cancellation sets. . . . 81
3.41 Rough aggregated workflow for the services in Figure 3.39. . . . . . 82
3.42 Final aggregated workflow for the services in Figure 3.39. . . . . . 83
3.43 Deploying the aggregation technique as a BPEL process. . . . . . . 88

4.1 Clothing Shop service selling shirts and trousers. . . . . . . . . . . 102
4.2 Clothing Shop workflow with places/conditions. . . . . . . . . . . 104
4.3 RG for the Clothing Shop service. . . . . . . . . . . . . . . . . . . . 104
4.4 Query workflow, RG and TT. . . . . . . . . . . . . . . . . . . . . . . . 106
4.5 MG for the example in Subsection 4.1.1. . . . . . . . . . . . . . . . . 106
4.6 Workflow of the Trousers Boutique service. . . . . . . . . . . . . 108
4.7 Phases of the adaptation process. . . . . . . . . . . . . . . . . . . . . 113
4.8 YAWL workflows for the CC and ME services. . . . . . . . . . . 115
4.9 SETs for the CC and ME services. . . . . . . . . . . . . . . . . . . . 117
4.10 Dual SETs of CC and ME, and the adapter’s SET. . . . . . . . . . 118
4.11 YAWL workflow of an adapter for CC and ME. . . . . . . . . . . 122

5.1 The Basic Pattern Template. . . . . . . . . . . . . . . . . . . . . . . . . 138
5.2 The Empty pattern template. . . . . . . . . . . . . . . . . . . . . . . . 143
5.3 The Receive pattern template. . . . . . . . . . . . . . . . . . . . . . . 144
5.4 The Reply pattern template. . . . . . . . . . . . . . . . . . . . . . . . 146
5.5 The Invoke pattern templates. . . . . . . . . . . . . . . . . . . . . . . 147
5.6 The Wait pattern template. . . . . . . . . . . . . . . . . . . . . . . . . 148
5.7 The Throw pattern template. . . . . . . . . . . . . . . . . . . . . . . . 149
5.8 The Terminate pattern template. . . . . . . . . . . . . . . . . . . . . 150
5.9 The Sequence pattern template. . . . . . . . . . . . . . . . . . . . . 153
5.10 The Switch pattern template. . . . . . . . . . . . . . . . . . . . . . . 154
5.11 The Flow pattern template. . . . . . . . . . . . . . . . . . . . . . . . 156
5.12 The Pick pattern template. . . . . . . . . . . . . . . . . . . . . . . . . 157
5.13 The Pattern Templates for the activities on the pick’s branches. . . 158
5.14 The While pattern template. . . . . . . . . . . . . . . . . . . . . . . . 160
5.15 The Assign pattern template. . . . . . . . . . . . . . . . . . . . . . . 162
5.16 High level view of the Scope pattern template. . . . . . . . . . . 164
5.17 The Pattern Template translating the scope’s activity. . . . . . . . 165
5.18 The FaultHandler pattern template. . . . . . . . . . . . . . . . . . . 167
5.19 The BeginFaultHandler task of Begin(FaultHandler). . . . . . . 167
5.20 The Pattern Template translating a catch/catchAll activity. . . . . . . 169
5.21 The EventHandler pattern template. ................................. 171
5.22 The Begin(EventHandler) pattern template. ........................ 172
5.23 The CompensationHandler pattern template. ....................... 173
5.24 The Compensate pattern template. ................................. 175
5.25 The Process pattern template. ................................. 177
5.26 High-level view of the GCD workflow. ................................. 185
5.27 Detailed view of the GCD workflow. ................................. 186
Chapter 1

Introduction

In this Chapter we first introduce the main ingredients of Web services and we briefly discuss some of their current limitations. Further on, we give a bird’s-eye view of the main objectives and contributions of this thesis. The Chapter ends by describing the structure of the thesis.

1.1 Context and Motivations

Modern software applications are dynamic, heterogeneous, distributed and heavily interacting. During the last decades, software development has changed a lot, from structured programming to object-oriented and to component based software engineering, and recently to service-oriented computing.

1.1.1 Web Services: Main Ingredients

Service-oriented computing [74] is emerging as a new promising computing paradigm that centres on the notion of service as the fundamental element for developing future distributed heterogeneous software applications. The W3C defines a Web service as “a software system designed to support interoperable machine-to-machine interaction over a network” [93]. In other words, Web services are “self-contained modular business applications that have open, Internet-oriented standards-based interfaces. They are distributed loosely coupled services providing business processes that can be accessed by customers and suppliers in a hardware, operating system and programming independent manner” [30].

Web services\(^1\) are dealing with interoperability and platform-neutral communications, in contrast to COM [67], EJB [68], and CORBA [69], which support platform interoperability but use their own component model and binary wire format for distributed communication. One of the main ingredients of Web services is the use of the Extended Markup Language (XML) [94] for building a standardised and

\(^1\)We shall use “service” and “Web service” interchangeably throughout the thesis.
platform-neutral syntax for sending data, and a way of defining and verifying the structure of the data being sent. Some of the benefits that we expect from using Web services are: interoperability, platform-agnosticity, ubiquity, investment protection, and component reuse.

The Web service model is heavily supported by a large group of organisations and standards bodies into defining a service-oriented view of computation having three component roles: the service user, the service provider, and the service registry. The provider exposes its service to users through a standards-based interface and a description of the service. The user finds, accesses, and interacts with the respective service in a standard way. Finally, the registry provides a location for publishing and locating Web services.

The three standardised pillars of the Web services platform are the Simple Object Access Protocol (SOAP), the Web Service Description Language (WSDL), and the Universal Description Discovery and Integration (UDDI).

SOAP [95] is a lightweight and loosely coupled protocol for the exchange of information in a decentralised and distributed environment. SOAP defines the way in which XML messages can be wrapped into envelopes in order to allow for the exchange of any kind of XML information. Furthermore, SOAP is protocol, language, platform, and operating system independent.

WSDL [100] is an XML language used to describe and locate Web services. A WSDL document describes the functionality of a Web service and it specifies how to access the respective service in terms of e.g., service endpoints, binding protocol, message format, or parameter names and types. Through WSDL documents an application provider can hide implementation details, while a service requester can abstract the platform dependent details. WSDL can define SOAP messages used to access Web Services, the protocols over which such SOAP messages can be exchanged, as well as the Internet locations from where these Web Services can be accessed.

UDDI [29] describes an online electronic registry for service descriptions where providers register themselves together with the services they offer, and from where requesters find such information. In other words, UDDI provides a model for publishing, validating, and invoking information about Web services. UDDI makes use of various taxonomies for categorising Web services such as the North American Industry Classification System (NAICS), or the Universal Standard Products and Services Classification (UNSPSC). The information registered in UDDI registries can be used to perform several types of searches: “White pages search” (using address, contact, and known identifiers), “Yellow pages search” (using taxonomies such as NAICS or UNSPSC), or “Green pages search” (using technical information about Web services).
1.1.2 Web Services: Some Open Issues

Two prominent challenges involved in the development of next generation software applications can be roughly synthesised as:

- **Aggregating** services to build a needed business application, and
- **Adapting** services to overcome mismatches in their interaction.

Service aggregation is an activity where a business entity interacts with a variety of service providers to re-brand, host, or offer a composition of services to its customers. Simply stated, service aggregation deals with building new (complex) services from (simple) existing ones. Two main advantages one obtains from aggregating services are the reduced application development times and costs, and the possibility to satisfy complex service queries (viz., requests). For example, a service developer can create a *Trip Planner* application by composing a *Flight Reservation* service with another *Hotel Booking* service. Furthermore, the newly created *Trip Planner* application can be used to fulfil user requests asking for applications that allow the booking of both flight tickets and hotel rooms.

The need for service adaptation comes from the dynamic, distributed and evolving nature of the Web services. On the one hand, service providers may wish to deploy existing services to new clients, or vice versa, without having to re-implement the core application code of the business application. Another important motivation comes from the need to wrap application code that cannot be modified such as legacy systems (e.g., systems to handle customers' accounts in banks) to meet new business demands. On the other hand, developers may face the problem that the services they wish to use for the generation of a new application do not interoperate successfully due to signature, semantic, or behaviour mismatches. A widely adopted solution to tackle these issues is through a disciplined use of adapters as services “in-the-middle” capable of mediating the information exchanged by the involved parties (e.g., [9, 27, 50]).

Currently, WSDL interfaces do not include any protocol information, that is, the order in which the operations are to be invoked. As a consequence, aggregating services only on the basis of their WSDL interfaces may lead to composite services that fail or lock. For example, a client of a *BookShop* service that exposes a WSDL interface defining *add2ShoppingCart* and *login* operations has to invoke the *login* operation before invoking the *add2ShoppingCart*, otherwise the interaction with the *BookShop* does not succeed.

Furthermore, the WSDL interfaces corresponding to interacting Web services may present signature mismatches (e.g., different operation names, or syntactic differences among the exchanged messages, such as different orderings of the message parts). For example, the code of a client wishing to interact with the *BookShop* service might include an *add2Cart* operation invocation instead of an *add2ShoppingCart* one. Although signature mismatches can be solved e.g., by calling the proper op-
eration in the adapter, the lack of protocol information hinders the adaptation of services that present behaviour mismatches.

Several approaches have emerged to enhance service descriptions with behaviour information [13, 71, 92, 99]. The most widely adopted is the Business Process Execution Language (BPEL) [13], which describes business processes through the specification of control and data logic around a set of (WSDL) Web service interactions. However, manual (BPEL-based) service aggregation is time consuming and error-prone, in that the responsibility of finding and correctly composing the appropriate services falls on the (human) designer.

Furthermore, equipping services with a description of their interaction behaviour does not per se eliminate the possibility of mismatches among service protocols. For example, the interaction between the BookShop service and a BookBuyer service could dead-lock if the former waits for a delivery address to be provided by the latter (e.g., through the invocation of a setDeliveryAddress operation of the BookShop), while the latter waits for the price of the book being purchased from the former (through the invocation of a setPrice operation of the BookBuyer).

Most of the current approaches to express Web service protocols lack formal semantics, and this hinders the formal verification of the consistency and validity of the languages, and of Web services’ properties such as lock-freedom. For example, the BPEL specification does not define the execution semantics of a business process in which an installed compensation handler is invoked more than once. Furthermore, software developers can create complex business applications through BPEL processes that orchestrate the operations offered by various Web services. However, a formal analysis is needed in order to check whether the composite can be executed successfully, e.g., without violating the order in which the operations of the participant services are to be invoked.

Another open problem is that behaviour-aware Web service aggregation and adaptation cannot be automatically employed to create heterogeneous Web services. Indeed, the current lack of a standard to describe the service behaviour allows for heterogeneous descriptions, but tools for the automated translation of service protocols are not yet available. Most service description languages use WSDL for describing the signature of services and SOAP for message exchanges. However, if the behaviour of the services to be composed is described using different protocol languages, the protocol of the composite cannot be automatically derived from those of the participant services. Furthermore, the compatibility of the interaction protocols of the services involved in the composition cannot be automatically verified since the protocols are expressed through different languages, among which there are no translations available.

Another issue towards achieving automated service aggregation and adaptation is due to the pure syntactic nature of WSDL service descriptions. Providers publish WSDL advertisements to UDDI registries, which in turn provide clients with keyword- or taxonomy-based service discovery capabilities. WSDL descriptions do not include any ontology information, and hence they are not “self-described” in a
1.1. CONTEXT AND MOTIVATIONS

machine-interpretable way. This severely limits the quality of the discovery results as the matched services may not necessarily offer the requested functionality, and hence fully-automated service aggregation and adaptation becomes unfeasible.

Moreover, service aggregation and adaptation often require to construct the dataflow of the composite or adapter service. In order to do so, one has to match the outputs of one service with the inputs of another one. For WSDL services this can be done automatically only on the ground of the names and/or types of the messages (or of the message parts) exchanged by the services. For example, the add2ShoppingCart operation of the BookShop service could have a book input message of a bookType type. However, the BookBuyer might be described as a BPEL process that invokes the add2ShoppingCart operation (of the BookShop) providing a scientificBookName message whose type is scientificBookType. Still, this mismatch could be overcome through the use of ontologies and ontology-based matching (e.g., [73]) if, for instance, scientificBookType and bookType are concepts in an ontology of books, where the former is a subtype of the latter. In this perspective, the Semantic Web initiative argues for the use of ontology languages such as OWL-S [71], WSDL-S [4], METEOR-S [66], SWSO [81], or WSMO [103] to enhance service descriptions so as to pave the way for the automation of the discovery, composition, execution, and monitoring of Web services.

As we already mentioned, the most widely adopted approach to service aggregation consists of manually generating BPEL processes from WSDL services. The semi-automated composition of services (e.g., [53, 64]) usually involves a service composition system that interacts with the requester in an iterative manner in order to obtain information about the requested service, and to construct aggregate service(s) out of the registered ones. Other approaches aim at offering platforms for composing services to achieve a desired goal, which may be expressed, for example, in terms of properties of the aggregate. Most of the fully-automated approaches to service aggregation employ artificial intelligence techniques such as planning (e.g., [11, 63, 84, 104]), still the goal is difficult to represent and the aggregation process is quite time-consuming. Moreover, to the best of our knowledge, existing techniques do not provide means to compose services described with different service description languages.

Several approaches tackle service adaptation at various levels of the Web services stack [74]. For example, [42, 82] address issues due to syntactic differences among the exchanged messages, [40, 78] mediate semantic mismatches among the exchanged messages, while [10, 14, 50] handle the aggregation of services that present mismatches in their communicating protocols. However, Web service adaptation is in its early stages and current approaches feature only partial solutions to the issues of adaptation.
1.2 Main Contributions of the Thesis

The ultimate objective of this thesis is to define a methodology for developing (Web) service aggregation and adaptation middleware, capable of suitably overcoming semantic and behaviour mismatches in view of application integration within and across organisational boundaries.

In order to tackle the above mentioned issues, we argue for the usage of *service contracts* [22] to describe service interfaces. Service contracts consist of *signature* (expressed through WSDL), *ontology information* (described with OWL [62], for example), and a description of the service *behaviour* through an abstract formal language (expressed by a YAWL [87] workflow). We argue that service contracts are a key ingredient for automating the processes of locating, aggregating and adapting Web services. On the one hand, the signature and ontology information can be exploited to overcome signature mismatches, as well as for matching service inputs and outputs so as to improve the process of locating services, or to tackle semantic mismatches. On the other hand, the protocol information is needed to tackle behaviour mismatches, and the generation of service compositions. Furthermore, the abstract formal language can be used both as a “lingua-franca” for translating the behaviour of real-world service descriptions, and as a basis for the formal analysis of the service’s behaviour. In this way one can check, for example, whether the composite or the adapted service may lock.

The three main contributions of this thesis can be roughly summarised as follows:

1. The definition of a Web service aggregation technique that, given a registry of (advertised) service contracts and a client service contract, automatically generates compositions of contracts that satisfy the client request. In short, by inspecting the execution traces of the services in the registry, we first individualize candidate sets of contracts that could satisfy the query collectively. For each candidate set, we generate the contract of the aggregate by suitably building its control- and data-flow, and we verify whether it actually complies with the request. The control-flow of the aggregate is obtained from the initial control-flow of the services in a candidate set, and from data-flow constraints obtained by matching the parameters of the services in the candidate set. A comparison of the traces generated by the composite service and by the query establishes whether the former satisfies the latter.

2. The definition of a Web service adaptation technique that can be used to tailor services both, to functional (viz., expressed in terms of inputs and outputs only), and behavioural (viz., expressed in terms of a workflow) client requests. Roughly, the first step of the core adaptation matches candidate (composite) services by checking the compatibility of their execution traces with respect

---

2Note that we use the term *contract* to denote a “rich service description” (e.g., as in [65]) and not “an agreement among multiple parties” (e.g., Web Service Level Agreements [102]).
to the given query. The second step generates (whenever possible) for each candidate service the contract of the service adapted so as to fulfil the query. For functional client requests, the workflow of a candidate service is modified so as to specify a refined behaviour of the initial service that enforces the needed adaptation. For behavioural client requests, the adaptation derives the service execution tree and the workflow of an adapter from the service execution trees of the candidate and of the client workflows. Should the interaction of the adapted service with the client one have at least one lock-free trace, we say that the adaptation is (partially/fully) successful. In particular, we will show how the adaptation can be employed for the automated generation of BPEL adapter processes, which allow two communicating BPEL processes whose interaction may lock to successfully interoperate.

We would like to stress out the fact that our service aggregation and adaptation techniques can be successfully unified since both processes rely on (input and output) service contracts. On the one hand, service aggregation may require some adaptation. In this case, the adaptation can be plugged with the aggregation for the customisation of services that do not (fully) satisfy client requests. Furthermore, service adapters and/or adapted services can be located and then composed with other services. On the other hand, the (behavioural) adaptation process discussed in this thesis constructs the contract of the adapted service as the aggregation between the original service and the generated adapter.

3. The definition of a translator of BPEL processes into YAWL workflows, as a contribution towards the automated generation of service contracts from real-world service descriptions. In this way, we pave the way for the automated aggregation and adaptation of BPEL processes, since the aggregation and adaptation techniques (points 1 and 2 above) make use of YAWL workflows to represent service protocols. Basically, we define a YAWL pattern for each BPEL activity, as well as for the whole BPEL process. The translator is compositional, in that the patterns of the BPEL structured activities are obtained by composing the patterns of their children activities. The role of patterns is twofold – they provide a unique representation of activities, and they provide an execution context for them. Given a BPEL process, the translator automatically generates its YAWL translation by first instantiating the pattern of each activity defined in the BPEL process, and then by suitably interconnecting the obtained patterns into the final workflow. Consequently, we argue that the translator also provides a light-weight semantics of BPEL processes.
1.3 Structure of the Thesis

The three main contributions of the thesis are described in Chapters 3, 4, and 5. In order to make the thesis self-contained, we provide an introduction to YAWL and to BPEL (in Chapters 2 and 5, respectively). Moreover, state-of-the-art and related work will be discussed separately in each Chapter so as to simplify the reading and to provide more focussed comparisons.

Chapter 2 provides some insights on the key elements and features of YAWL by means of a few simple workflows.

Chapter 3 describes the aggregation of Web services. Section 3.1 motivates the need to aggregate services for answering complex client queries. In Section 3.2 we briefly describe the main aggregation phases, while Section 3.3 informally introduces the service contracts. Section 3.4 first describes the process of generating the execution traces of a service (Subsection 3.4.2) through a reachability analysis (Subsection 3.4.1). Furthermore, it presents the process of matching advertised service contracts (Subsection 3.4.3), their composition into aggregates (Subsection 3.4.4), and the process of verifying whether the aggregates fulfil the client request (Subsection 3.4.5), as well as an informal proof of the correctness of the aggregation process (Subsection 3.4.6).

Some more illustrative examples showing how the aggregation process copes with YAWL conditions, composite and multiple instance tasks, as well as cancellation sets are given in Section 3.5. In Section 3.6 we briefly describe Sator, the proof-of-concept prototype implementation of the core aggregation. Furthermore, Section 3.7 analyses the complexity of our approach.

Section 3.8 is devoted to discuss the main middleware aspects regarding the deployment of the aggregation phases and of the entire aggregation process as Web services. Finally, in Section 3.9 we briefly review the related work, while in Section 3.10 we present some concluding remarks.

Chapter 4 presents the adaptation of Web services. On the one hand, Section 4.1 focuses on the customisation of services to functional client requests (viz., functional adaptation). The Section starts with the presentation of a simple motivating example (Subsection 4.1.1). Furthermore, Subsection 4.1.2 briefly recalls how the methodology matches a candidate service, and how it verifies whether the candidate fulfils the query. The following Subsection 4.1.3 presents the core of the functional adaptation process, that is, the generation of the adapted service contract. In Subsection 4.1.4 we informally prove the correctness of the functional adaptation technique.

On the other hand, in Section 4.2 we describe the technique of adapting services to behavioural client queries (viz., behavioural adaptation). We illustrate this form of adaptation by showing how to generate BPEL adapters that allow two interacting business processes presenting behavioural mismatches to interact successfully. Subsection 4.2.1 introduces a simple motivating example, while Subsection 4.2.2
1.3. STRUCTURE OF THE THESIS

describes the main phases of the adaptation, with a focus on the generation of the adapter, and validation of the adapted service, while including bird’s-eye views of the process of service translation and adapter deployment. Subsection 4.2.3 informally proves the correctness of the behavioural adaptation technique.

Finally, Section 4.3 analyses the complexity of the two adaptation techniques, Section 4.4 briefly discusses related work, while Section 4.5 presents some concluding remarks.

Chapter 5 thoroughly describes the specification of a translator of BPEL processes into YAWL workflows (BPEL2YAWL for short). Section 5.1 briefly introduces BPEL. Section 5.2 is devoted to the specification of the BPEL2YAWL translator. In Subsections 5.2.1 and 5.2.2 we define the patterns used for translating the BPEL basic and structured activities, respectively, while Subsection 5.2.3 describes the translation of BPEL processes. Section 5.3 thoroughly presents a simple translation example, while Section 5.4 briefly reviews related work. Finally, Section 5.5 provides some concluding remarks.

Chapter 6 briefly reviews the main contributions of the thesis and draws some final concluding remarks.
Chapter 2

Background: YAWL (Yet Another Workflow Language)

In this Chapter we briefly introduce the key elements and features of YAWL [87], and we argue that YAWL is a promising candidate to be used as an abstract workflow language for describing service behaviour. Further insights on the constructs and usage of YAWL will be given in the following Chapters.

YAWL is a new proposal of a workflow/business processing system, that supports a concise and powerful workflow language and handles complex data, transformations and Web service integration. YAWL defines twenty most used workflow patterns gathered by a thorough analysis of a number of languages supported by workflow management systems. These workflow patterns are divided in six groups (basic control-flow, advanced branching and synchronisation, structural, multiple instances, state-based, and cancellation). A detailed description of them may be found in [88].

YAWL extends Petri nets by introducing some workflow patterns (for multiple instances, complex synchronisations, and cancellation) that are not easy to express using (high-level) Petri nets. Being built on Petri nets, YAWL is an easy to understand and to use formalism. Furthermore, Petri net based tools such as [91], and YAWL-based tools such as [90] can be employed to formally analyse YAWL workflows. Furthermore, in Chapter 3 we provide some insights on how reachability graphs and modified reachability trees can be employed to check formal properties of YAWL workflows such as, lock-freedom.

With respect to process algebras, YAWL features an intuitive (graphical) representation of services through workflow patterns. Furthermore, as illustrated in [85], it is likely that a simple workflow which is troublesome to model for instance in π-calculus may be instead straightforwardly modelled with YAWL. A thorough comparison of workflow modelling with Petri nets versus π-calculus may be found in [85]. With respect to the other workflow languages (mainly proposed by industry), YAWL relies on a well-defined formal semantics. Moreover, not being a commercial language, YAWL supporting tools (editor, engine) are freely available. With respect to
state-based models, YAWL can be successfully employed to represent complex service behaviour (e.g., fault, event and compensation handlers, and synchronisation in BPEL processes) and the data-flow of services. For example, YAWL employs data-flow aware predicates to determine the control-flow (see the File Server example hereafter). In Chapter 3 we exploit YAWL predicates for computing the conditions in which client requests can be satisfied. Since both BPEL and YAWL are XML-based languages that use XPath for data management, one can encode the data-flow of BPEL processes into YAWL workflows, and vice versa. For example, the translator of BPEL processes into YAWL workflows that we introduce in Chapter 5 (BPEL2YAWL for short) defines a YAWL pattern for each BPEL activity. Then, the transition conditions associated to outgoing synchronisation links of BPEL activities can be represented in the translated workflows as YAWL predicates on links outgoing from the translated patterns. Furthermore, although the aggregation and adaptation methodology we define in this thesis copes with all YAWL constructs, all the YAWL patterns defined by the BPEL2YAWL translator do not employ the YAWL OR-join, whose complex semantics may lead to thorny workflow analyses. Last but not least, note that YAWL can be used to represent partial service behaviour, e.g., the communication behaviour of a service with (some of) its partners.

Figure 2.1 graphically depicts some examples of YAWL workflows. Search Engine is a workflow that consists of an input condition, two tasks, File Info and Download URL, and an output condition, all linked in a sequence. YAWL conditions and tasks can be interpreted as Petri net places and transitions, respectively [87]. Hence, the execution of the workflow starts by placing a token in its input condition. As a consequence, the File Info task becomes enabled and ready to be executed. Its execution requires two values for its input parameters fName and os.1 The workflow continues with the execution of the Download URL task, as YAWL considers implicit conditions for tasks that are linked directly. Download URL outputs a value and places a token in the output condition of the workflow, which marks the termination of the Search Engine workflow.

The File Downloader workflow consists of one task only, which inputs a URI and outputs a binaryData value. The workflow starts by placing a token in its input condition. The Download task becomes then enabled, and its execution marks the termination of the workflow by placing a token in the output condition.

Another example is the File Server workflow, which starts by executing the Get Filename task. The workflow continues next with either Locate URI, or with Send File. The decision is made by the XOR-split control construct of the Get Filename tasks which places a token in only one of its output links. YAWL uses predicates to determine the control-flow in case of XOR- and OR-splits. For example, a token is sent to Locate URI if and only if the predicate limitedBandwidth is true,
or if both predicates limitedBandwidth and Cached(Filename) are false, because limitedBandwidth is the default predicate. Cache File needs one token only for being enabled due to its XOR-join. Its execution finishes the workflow as a token is placed in the output condition.

Finally, the Fetch Application workflow consists of three tasks executed in a sequence. While the execution of the first two tasks is not constrained by any input parameters, its third task, Get File, requires a dataFile value as input.

From a control-flow perspective, a YAWL file describes a workflow specification that consists of one or more extended workflow nets (or EWF-nets for short) arranged in a tree-like structure. For instance, all workflows in Figure 2.1 consist of one EWF-net only. An EWF-net is a graph where nodes are tasks or conditions, and arrows define the control-flow relation. Each EWF-net has a single input condition and a single output condition. For example, all the workflow specifications depicted in Figure 2.1 consist of a single EWF-net. A YAWL task may be either atomic or composite. An atomic task (e.g., FileInfo or GetFilename) corresponds to a leaf of the tree. A composite task corresponds to an EWF-net at a lower level in the hierarchy. The EWF-net without any composite tasks referring to it is called top-level workflow and it corresponds to the root of the tree-like hierarchy. A task can have multiple instances which can be created either statically or dynamically. Lower and upper bounds are used to specify the number of instances that can be created. Furthermore, a threshold value may be used to indicate the number of sufficient instances that have to complete in order for the task to terminate.

A task Q is to be executed after another task P if there is an arrow from P to Q. For example, the Download URL task of the Search Engine workflow (see Figure 2.1) can be executed only after executing the File Info task. Tasks employ one join and one split construct. A join or split control construct may be one of the following: AND, OR, XOR, or EMPTY. Intuitively, the join specifies “how many” tasks before P are to be terminated in order to execute P, while the split construct specifies “how many” tasks following P are to be executed. The EMPTY-join (split) is used when only one task execution precedes (follows, respectively) the execution of P. For instance, the Download task of the File Downloader workflow employs EMPTY-join and -split constructs. Furthermore, the Get Filename and Cache File tasks of the File Server workflow employ, respectively, a XOR-split and a XOR-join. YAWL tasks may also be connected directly one another (i.e., without an in-between condition) and in this case one may assume an implicit (empty) condition between them.

YAWL uses predicates in the form of logical expressions to express the control-flow in the case of XOR- and OR-splits. On the one hand, tokens are placed into places by firing tasks depending on their split constructs and on the YAWL predicates (if present). For tasks with EMPTY- (AND-) splits, YAWL considers implicit (empty) conditions and a token is generated for (all) the output place(s). In the case of XOR- or OR-splits, YAWL uses predicates to determine which output places will receive tokens. All predicates of such a split are ordered (by the workflow designer)
and one is chosen as default (with lowest order). One may see in Figure 2.1 that the Get Filename task of the File Server workflow has two predicates – “limitedBandwidth” (which is the default one), and “Cached(FileName)”. For a XOR-split, a token flows along the link corresponding to the predicate with the lowest order that evaluates to true. For an OR-split, a token is sent along all links whose predicates evaluate to true. For both split constructs, if all predicates are false then a token is sent along the default link only.

On the other hand, places are used to enable tasks for execution. If the task has an EMPTY-join then its input place has to contain a token for the task to be enabled. For an AND-join, all input places have to contain tokens. In the case of a XOR-join at least one input place has to have a token. Hence, the Cache File task of the File Server workflow can be executed provided either Locate URI, or Send File was executed. Finally, according to [87], a task having an OR-join is enabled only when at least one of its input places contains a token and no other tokens can be placed in its remaining (empty) input places.

Another feature of YAWL is that a task may have a cancellation set associated to it. The cancellation set consists of conditions and tasks. When a task is executed, all tokens from its cancellation set (if any) are removed. Cancellation sets are useful, for example, to prevent tasks from being executed given some particular circumstances, or even to terminate the execution of the entire workflow. For example, if multiple tasks are used to book each one flight ticket with a different airline company, the first one to be executed successfully should inhibit the other ones, while the impossibility to book a flight ticket with any company should immediately terminate the workflow, without having to rent a car.

Hence, we argue that YAWL can be used to define complex service control-flow. For example, YAWL supports the sequential and parallel execution of tasks. The former corresponds to simply linking tasks in a sequence. The latter can be defined using a task with an AND-split as initiator of the flow (call it Begin), and another task with an AND-join marking the termination of the flow (call it End). Furthermore, the tasks in the flow simply have to be connected as output of the Begin task, and as input of the End task. By doing so, the execution of the Begin will enable all tasks in the flow, while the End task will be enabled only after the reception of tokens from all of the task in the flow. Deterministic choices (e.g., useful to implement conditional or iterative behaviour like the BPEL switch or while) can be implemented by tasks with XOR-splits. Furthermore, non-deterministic choices (e.g., the BPEL pick operation) can be achieved through the use of deferred choice constructs, consisting of a condition taken as input by several tasks. In this way, the environment (viz., the invoker of the workflow) decides the control-flow by choosing one task to be executed. Consequently, the respective task consumes the token in the input condition, and hence, the other tasks cannot be further executed.

Furthermore, from a data-flow perspective, YAWL uses XMLSchema, XPath and XQuery for dealing with data. Variables are defined at both EWF-net and task levels, and bindings between them are realised through XQuery expressions.
Therefore, we argue that YAWL is a good candidate to model the data-flow of the various existing service description languages (e.g., BPEL [13], OWL-S [71], WSCDL [99]), since all are XML-based languages and manipulate data in a similar way. For example, BPEL assignments could be implemented by suitable mappings among the inputs and outputs of YAWL tasks and/or of EWF-net variables.

In Chapter 5 we describe a pattern-based compositional translator of BPEL processes into YAWL workflows. We show how YAWL can be employed to generate workflows that model the complex behaviour of BPEL, including synchronisation links, fault, event, and (explicit) compensation handling.

In the following we shall use the terms workflow and service interchangeably, due to the usage of YAWL workflows to model the behaviour of Web services.
Figure 2.1: Examples of YAWL workflows.
Chapter 3

Web Service Aggregation

In this Chapter we present a Web service aggregation approach\(^1\) that, given a registry
of (advertised) service contracts and a client service contract, automatically generates compositions of contracts that satisfy the client request. The main assumption we make in this thesis is the existence of a registry of service contracts. We reckon that this is a fair assumption since, e.g., we will show in Chapter 5 how BPEL processes can be automatically translated into YAWL workflows, while the ontology information can be (at least partially) generated from semantically enhanced WSDL descriptions (e.g., WSDL-S [4] or OWL-S [71] service descriptions).

Basically, the purpose of our aggregation technique is to construct a composite service that successfully interacts with, and provides all the needed inputs for the successful execution of the client service. For instance, consider the set of services illustrated in Figure 2.1 containing three different services whose inputs and outputs (IOs) can be suitably related so as to build an aggregated service. The three services are *Fetch Application*, *Search Engine*, and *File Downloader*.

Recall that the *Fetch Application* service is in charge of downloading software applications. It first outputs the name and the support platform of the desired application and then it waits to receive the application file. Its successful execution is conditioned by the reception of the application file, which could be obtained, for example, from the *File Downloader* service that inputs the URI of the file to be downloaded and outputs the respective file. Although the *File Downloader* can provide the value needed for the successful execution of the *Fetch Application* service, its execution is in turn constrained by the availability of the application’s URI. Again, we can assume that such a URI can be obtained through the execution of the *Search Engine* service, which first takes as input the name and the platform of a desired application, and then it outputs the URI from where the respective application can be downloaded. At this point, one may note that a successful aggregation of the three services can be obtained, as *Fetch Application* outputs the name and the platform of the desired application, inputted next by the *Search Engine*, which

---

\(^1\)Some of the results described in this Chapter have been preliminary reported in [19, 22, 25].
outputs the download URI of the application. Then, the *File Downloader* inputs the URI and outputs the binary file, which is taken as input by the *Fetch Application*.

In Section 3.1 we will show how it is possible to generate successful aggregations of the services in Figure 2.1, if we assume that the methodology takes as input the *Fetch Application* client service contract, together with a registry containing the contracts of the *Search Engine, File Downloader, and File Server* services.

The core of the aggregation technique consists of three main phases:

1. In a first phase we individuate candidate sets of services that may be aggregated to satisfy the client service. Roughly, a candidate set is characterised by the fact that its services can provide collectively all the inputs needed by the client service, and vice versa.

2. In a second phase, for each candidate set we generate the contract of the composite service, by first performing a control-flow and then an (ontology-aware) data-flow analysis of the behaviour of the contracts to be aggregated. The former generates the main control-flow of the aggregate from the individual control-flow of each service, while the latter generates additional control-flow dependencies based on the matches among the IOs of the participant services. The result is a YAWL workflow that expresses the interplay among the aggregated services, namely all the control-flow and data-flow relationships among them.

3. Finally, the third phase verifies for each aggregated contract whether its interaction with the client is lock-free, and whether it actually fulfils the request.

To the best of our knowledge our aggregation technique is the first one to offer all of the following features:

- It is a fully-automated approach capable of generating service aggregations that fully/partially satisfy behavioural queries,
- It supports both service selection and aggregation at the level of traces (and not simply at the entire service level),
- It is efficient to amenable implementations as it relies on service contracts and execution traces, which can be both generated off-line, and
- It can be exploited to locate and aggregate services written in different languages, and to generate multiple deployments of the aggregated contract, given that it relies on intermediate YAWL descriptions of the behaviour of services, as well as
- It can be used to locate, aggregate and adapt BPEL processes, as it straightforwardly integrates with the adaptation approach described in Chapter 4, and with the BPEL2YAWL translator presented in Chapter 5.
The Chapter is organised as follows. Section 3.1 motivates the need to aggregate services for answering complex client queries, using the services introduced in Chapter 2. In Section 3.2 we briefly describe the main aggregation phases, while Section 3.3 informally introduces the service contracts.

Section 3.4 is dedicated to presenting the aggregation approach. We start with the description of the reachability analysis (Subsection 3.4.1) and of the service execution traces (Subsection 3.4.2). Next, we describe the processes of matching advertised services (Subsection 3.4.3), the core aggregation and the generation of the aggregated contract (Subsection 3.4.4), and the contract validation phase (Subsection 3.4.5), as well as an informal proof of the correctness of the aggregation process (Subsection 3.4.6).

Some more illustrative examples showing how the aggregation process copes with YAWL conditions, composite and multiple instance tasks, as well as cancellation sets, are given in Section 3.5. In Section 3.6 we briefly describe Sator, the proof-of-concept prototype implementation of the core aggregation. Furthermore, Section 3.7 analyses the complexity of our approach. Although this thesis focuses on the methodological aspects of our aggregation approach, Section 3.8 is devoted to discuss the main middleware aspects regarding the deployment of the aggregation phases and of the entire aggregation process as Web services. Finally, in Section 3.9 we briefly review the related work, while in Section 3.10 we present some concluding remarks.

### 3.1 Motivating Example

Consider a client service, *Fetch Application*, whose YAWL behaviour is given in Figure 2.1, and suppose that the client wishes to use this service to download applications. Informally\(^2\), *Fetch Application* firstly outputs the name and the target platform of the desired application, and then it waits for the data file. Note that the execution of the *Fetch Application* workflow is constrained by obtaining a value for the input parameter `dataFile` of the *Get File* task.

Consider further a registry consisting of the three services whose behaviour are given by the YAWL workflows presented in the top part of Figure 3.18. *Search Engine* is a service that, provided a file name and a target operating system, outputs the URL address from where the respective file can be downloaded. *File Downloader* is a download accelerator service. It inputs the URI of a requested file and it outputs the file upon completion. *File Server* is a service offering the functionality of a search engine with caching capabilities. Firstly, it inputs the name of the file to be downloaded. If the available bandwidth does not permit a quality download, or if the file is not cached, the service outputs the URI of a similar file on a different server. Otherwise, it outputs the file. Finally, it caches the requested file.

\(^2\)We recall that the YAWL workflows in Figure 2.1 (together with the main YAWL constructs) have been described in Chapter 2.
As we shall see later, the dataFile input of the Fetch Application service can be obtained from (compositions of) the services in the registry. It is important to note that the example is not supposed to present a software masterpiece, as we would like to underline the fact that different services, written by different providers with different programming styles and backgrounds, may present interaction issues.

For simplicity, we shall consider exact matches [73] among the parameters\(^3\) of the previously mentioned services in each of the following sets: (a) \{ appName, fileName, fName \}, (b) \{ URL, URI \}, (c) \{ platform, os \}, and (d) \{ dataFile, file, binaryData \}.

Single service matching approaches based on IOs [73] would hence match only the File Server service, as the input requested by Fetch Application is generated by the File Server and dually, the inputs needed by File Server are to be given by the Fetch Application service. However, File Server can satisfy such request only if the requested application is cached and there are no bandwidth issues with the server.

Other IO-based matching approaches tackling the discovery of composite services satisfying a query would be able to individuate the sets of services that collectively satisfy the request. Two possible matches would be given by \{ Search Engine, File Downloader \}, and by \{ File Server, File Downloader \}. In the former, File Downloader provides the input file for the Fetch Application, yet it requires a URI, which can be obtained by executing the Search Engine service. Note that, in this case, the inputs of the Search Engine service are to be obtained from the outputs of the Fetch Application service. In the latter, the execution of the Send File task of the File Server workflow produces the input needed by Fetch Application. As in the previous case, the name of the file to be downloaded, that is needed for the execution of the File Server service, is to be given by the execution of the Set Name task of the Fetch Application workflow. However, since such approaches view both the client request and the advertised services as black-boxes (i.e., expressed in terms of (requested) inputs and (generated) outputs), their composition might lock. For example, the composition of File Server with File Downloader blocks if the file is cached by the former and if there are no bandwidth problems, because the former outputs the cached file instead of the URI, which is needed by the latter.

Many approaches to composition-oriented discovery of services [7, 8, 15, 16, 17, 45, 48, 53] take into account the behaviour of the services in the registry in order to look for a composition of them able to satisfy a black-box request. However, they do not deal with behavioural queries for which the IOs are requested/offered at various execution steps of the client service. As a consequence, the aggregation between the composite service generated by the matching approach and the client request might lock once again.

The aggregation approach we describe in this Chapter looks for compositions of

\(^3\)Note that, in order to ease further the description of the methodology, by “parameters” we refer either to the names of the parameters, or to their corresponding ontology concepts, depending on the context.
advertised services that satisfy the client service. In the following, we will show how one may obtain three possible scenarios for satisfying the Fetch Application service by aggregating it with one of the following sets of services: (a) \{ Search Engine, File Downloader \}, or (b) \{ File Server \}, or (c) \{ File Server, File Downloader \}, and we will discuss and compare these three possible solutions.

### 3.2 Overview of the Aggregation Phases

The aggregation approach we propose can be synthesised by the following phases:

0. **Service Translation.** This preliminary phase deals with translating real-world descriptions (e.g., BPEL + ontology information, or OWL-S) of the services to be aggregated into equivalent service contracts using WSDL for the signature, YAWL as an abstract workflow language for expressing its behaviour, and OWL, for example, for expressing the ontology information. An analysis of how to transform BPEL specifications into workflow patterns can be found in [97]. Furthermore, in Chapter 5 we thoroughly describe a translator of BPEL processes into YAWL workflows. This phase may be done off-line and hence it is not a burden for the aggregation process.

1. **Service Matching.** This phase searches for candidate sets of service traces that together are able to satisfy a maximum number of traces of the client service. Each such candidate set together with the matched client traces form a “closed workflow” in the sense that the set of inputs needed by them collectively is included in the set of outputs generated by them. Still, one has to verify whether the services corresponding to traces in the candidate set may be successfully aggregated with the client one. This phase is also in charge of deriving a data-flow mapping among the services involved in the aggregation. The data-flow dependencies are obtained from matching workflow parameters, on the one hand, based on exact/subsumes/plug-in matches [73], and, on the other hand, by using sets of semantically equivalent parameter types given by the client. The latter allows us to cope with cross-ontology mapping. Hence, the service matching phase automatically generates the data-flow mapping by considering exact/subsumes/plug-in matches among parameter types in the same ontology, and by considering exact matches among the semantically equivalent parameter types that belong to (possibly) different ontologies.

2. **Core Aggregation and Contract Generation.** This phase is applied on each candidate set obtained at the previous phase, and it deals with generating the contract of the aggregated service. For each workflow to be aggregated, its YAWL tasks are expanded with explicit data- and control-flow (dummy) constructs, also called Input/Output Control/Data enabler tasks (or ICs/IDs/OCs/ODs for short). We then express the initial control-flow connections in terms of the newly added ICs and OCs. Using the data-flow mapping
obtained at the previous phase, we suitably link IDs and ODs of the added
dummies in order to construct the data-flow of the aggregate. In this way we
obtain the “rough” behaviour of the aggregated service. We then optimise it
by eliminating redundant dummies and control-flow constructs. The signature
and the ontological description of the aggregate are obtained from the union
of the signatures and ontological descriptions of the participant services. To-
gether with the previously obtained behaviour they form the service contract
of the aggregated service.

3. Contract Validation. For each aggregated contract we verify whether its
successful traces (viz., execution traces for which the service terminates suc-
cessfully) satisfy the previously matched successful traces of the client service.
Informally, for each matched successful trace of the client, we have to check
whether all tasks executed in this trace are executed in at least one successful
trace of the aggregate. The final result of the aggregation process is a list
of aggregated service contracts that fully/partially satisfy the request. The
output list is ordered according to the number of unconstrained successful
traces, where the constraints are given by the YAWL predicates deciding the
control-flow.

4. Service Deployment. Finally, the contract of a successfully aggregated ser-
dvice can be deployed as a real-world Web service (e.g., described using OWL-S,
or BPEL + ontology information). Clients will hence see the aggregation as
another Web service that can now be discovered and further aggregated with
other services. This phase is the “inverse” of the Service Translation phase.

3.3 Service Contracts

Currently, providers publish (purely syntactic) WSDL [100] advertisements to UDDI
[29] registries (constructed in the style of yellow pages) that in turn provide clients
with keyword- or taxonomy-based service discovery capabilities.

On the one hand, WSDL descriptions do not include any ontology information,
and hence they do not provide a machine-interpretable “self-description” of services.
This severely limits the quality of the discovery results, as the matched services may
not necessarily offer the requested functionality, and hence fully-automated service
discovery becomes unfeasible.

On the other hand, WSDL descriptions lack behaviour information. A direct
consequence of this is that service compositions may lock during execution. Stated
differently, without any protocol information (e.g., order of messages sent/received),
no guarantee on the behaviour of service compositions can be ensured.

Various proposals have been put forward in order to enhance service descrip-
tions. WSDL-S [4], OWL-S [71], SWSO [81], WSMO [103], or METEOR-S [79]
annotate services with ontology information. BPEL [13], WSCDL [99], METEOR-
S [3], OWL-S [71], SWSO [81], or recently YAWL [87] add protocol information to
3.3. SERVICE CONTRACTS

service descriptions. All the above proposals can be in principle exploited to improve the accuracy of service matching, to extend the properties of service compositions, as well as to automatise both processes.

The methodology described in this thesis aims at setting the basis for the development of Web service aggregation and adaptation middleware, capable of suitably composing and adapting services described using possibly different process/workflow modelling languages (e.g., BPEL [13], OWL-S [71], or WSCDL [99]), as well as of supporting multiple deployments of the aggregated or adapted contracts as real-world Web services. The difficulties of achieving this aim mainly arise from the fact that most of the languages that express the service behaviour lack ontology information and/or formal semantics. Furthermore, the current lack of a standard to describe the service behaviour allows for heterogeneous descriptions, but tools for the automated translation of service protocols are not yet available. Consequently, the automated creation of heterogeneous Web services is still an open challenge.

In order to tackle these issues, we consider services that are described by contracts [65], and we argue that contracts should in general include different types of information (see Figure 3.1):

- **Signature**,  
- **Ontology information**,  
- **Behaviour**, and optionally,  
- **Extra-functional properties**.

The signature can be expressed in terms of WSDL, which is the current standard for describing Web service interfaces. Following [71], we argue that (WSDL) signatures should be enriched with ontology information (e.g., expressed with OWL [62] or WSDL-S [4]) to better capture the semantics of services, and necessary to automatise the process of overcoming signature and semantic mismatches, as well as service selection, composition, and adaptation. Still, the information provided by the signature and ontology description levels is necessary but not sufficient to ensure a correct inter-operation of services.

Figure 3.1: Service contract levels.
A desired feature of our methodology is to translate the behaviour of real-world services into equivalent descriptions expressed through an abstract language with a well-defined formal semantics, and vice versa. The intermediate language should serve as a lingua franca for expressing the service behaviour. An immediate advantage of using such an abstract formal language is the possibility of developing formal analyses and transformations, independently of the different languages used by providers to describe the behaviour of their services.

We argue that a good trade-off between expressiveness and ease of verification of service contracts is to consider the behaviour of a Web service as modelling the interaction pattern, that is, the essential aspects of the finite interactive protocol (i.e., order of operations) that a service may present (repeatedly) to its environment. Hence, following [65], we argue that contracts should also expose a (possibly partial) description of the interaction protocols of services. Indeed, such information is necessary to ensure a correct inter-operation of services, e.g., to verify absence of locks.

As motivated in Chapter 2, we consider that YAWL [87] is a promising candidate to be used as an abstract formal language for describing the service behaviour. On the one hand, YAWL offers a common way to define complex control- and data-flow. On the other hand, the translation of Web service protocols into YAWL gives the possibility of formally analysing Web service properties through the use of Petri net and YAWL-based analysis tools (e.g., [90, 91]).

Finally, we argue that service contracts should expose, besides signature, ontology information, and behaviour, also so-called extra-functional properties, such as performance, reliability, or security. (We will not however consider these properties in this thesis, and leave their inclusion into the methodology as future work.)

### 3.4 Description of the Aggregation Approach

We start with the description of the reachability analysis (Subsection 3.4.1) and of the service execution traces (Subsection 3.4.2), in which we introduce some tools useful for the processes of service matching and analysis. Next, we describe the processes of matchmaking advertised services (Subsection 3.4.3), the core aggregation and the generation of the aggregated contract (Subsection 3.4.4) as well the contract validation phase (Subsection 3.4.5). Finally, we informally prove the correctness of the aggregation process (Subsection 3.4.6).

#### 3.4.1 Reachability Analysis

YAWL is a language built upon Petri nets (PNs) and hence the abundance of analysis tools for the latter could be employed for the analysis of YAWL workflows. For example, one might want to verify properties such as:
3.4. DESCRIPTION OF THE AGGREGATION APPROACH

- **safeness (k-boundness).** A PN is safe (k-bound) if any of its places does not contain more than one (k) token(s) under any circumstances.

- **conservativeness.** A PN is conservative if the total number of tokens in the net is constant.

- **reachability.** A PN marking is a vector of all the places in the PN, where each element in the vector holds the number of tokens in the respective place. A PN marking $M$ is reachable from another marking $M'$ if there exists a sequence of transitions that takes the PN from $M'$ to $M$.

- **coverability.** A PN marking $M$ covers another marking $M'$ if all transitions enabled by $M'$ are enabled by $M$ as well.

- **deadlock.** A PN marking $M$ reachable from an initial marking $M_0$ is in a deadlock if it enables no transitions.

- **liveness.** A PN transition is live if it can become firable from any reachable marking. Note that liveness implies deadlock freedom and not vice versa.

Since the introduction of the PNs, these issues were of a great concern for the researchers. The reachability tree (RT), or its representation as a reachability graph (RG) were introduced for the study of reachable markings. Consider the workflows in Figure 3.2 obtained from the workflows in Figure 2.1 by representing the implicit YAWL conditions between each two tasks.

Intuitively speaking, the RG of a YAWL workflow describes all its execution traces. Following [105] we derive a RG having markings as nodes and labelled arrows as edges. A marking $M$ consists of the set of all workflow places containing tokens and it is denoted as $C_1 + \ldots + C_j$. An arrow states that the workflow execution state may evolve from a marking $M$ into a marking $M'$ and it is labelled with the task that fires and – in the case of OR- and XOR-splits – also with the places that receive tokens. Figure 3.3 depicts the RGs corresponding to the four workflows in Figure 3.2. For example, the RG of the Search Engine workflow evolves from the initial marking $C_1$ into the marking labelled $C_2$ by executing the File Info task. Furthermore, the execution of the Get Filename task of the File Server workflow leads to a token being placed either in the place $C_7$, or in the place $C_8$.

The RG is incrementally built by starting from the initial marking, which contains the input condition only, and by looking for tasks that can be enabled. Labelled arrows and new markings are then incrementally added to the graph. Checking whether a task having an OR-join is enabled is done using the algorithm given in [105].

In the rest of the thesis we shall use the following terminology:

- **initial marking $M_i$:** the marking without incoming links. It contains only the initial condition of the workflow.
Figure 3.2: Motivating example workflows with explicit conditions.

- **final marking** $M_f$: the marking containing only the output condition of the workflow. It does not have outgoing links. Note that we consider one final marking only, which corresponds to a proper completion of the workflow [87].

- **execution trace** (or **trace** for short): a path originating in $M_i$ and ending in a marking $M$ of the RG.

- **successful execution trace**: an execution trace that ends in $M_f$.

- **deadlock**: an execution trace ending in a marking without outgoing links that is not $M_f$.

- **livelock**: an execution trace containing an infinite loop (hence not ending in $M_f$).

The main limitation of the RG is that it has an infinite number of markings for unbounded workflows, that is workflows with at least one place that can contain an infinite number of tokens (due to loops in the workflow). Karp and Miller [44] proposed the **finite reachability tree** (FRT) (or **coverability tree** (CT)) and its possible
representation as a coverability graph (CG) as a solution to representing the infinite space-state of unbounded PNs. The key feature of the FRT is the introduction of the ”-symbol to represent a place with a potentially infinite number of tokens in markings resulting from some transitions firing loops. A marking that contains at least one ”-symbol is called ”-marking. The construction of the FRT depends on the order in which the markings are considered and, in general, it is not minimal. (The minimal CT was proposed by Finkel [35] yet it is more computationally expensive.) The FRT can be used to determine properties such as safeness, boundness, conservativeness, and coverability. Furthermore, it can be used to determine the liveness of the PN when the tree contains no ”-markings (i.e., a finite tree). However, the FRT cannot be used to determine liveness, deadlock, or reachability due to the loss of information caused by the ”-symbol.

In order to tackle these properties, Wang et al. [96] formalised the modified reachability tree (MRT), which uses ”-numbers instead of ”-symbols. Similarly to FRTs, a MRT ”-marking contains at least one ”-number. ”-numbers denoted by \( k\omega_n + q \) are subsets of integers of the form \( \{ ik + q \mid i \geq n \} \), where the base \( k \in \mathbb{N}^+ \), and the least bound and respectively, the reminder \( n,q \in \mathbb{Z} \) and they can capture more information on the structure of the infiniteness than ”-symbols. For example, a place in a marking to which it corresponds a \( 2\omega_1 \) ”-number describes that the respective place holds an even number of tokens, not less than 2. The algorithm for building the MRT is a generalisation of the algorithm for building the FRT and, as the authors note, their complexities are similar.
We propose the usage of the MRT algorithm defined in [96]\(^4\) to build the MRT of a YAWL workflow with the purpose of analysing YAWL workflows, and consequently for the analysis of service behaviour. However, in order to keep the presentation manageable, we shall not go into any details about the construction of the MRT. Moreover, in the following we shall use the RG for the presentation of our methodology as:

- For bounded workflow nets, the MRT and the RT, which are the base of the RG, offer the same kind of information due to the fact that they both contain the same markings. All the examples employed in this Chapter have bounded representations. Moreover, our main concern in this thesis is the lock-freedom of the composite services. If the workflow is bounded (i.e., its RG representation is state-space finite), deadlocks can be seen in the RG as non-final markings without outgoing links. Furthermore,

- The RG provides a more compact and easier to follow representation than the MRT.

### 3.4.2 Service Execution Traces

We define the Trace Table (TT) of a workflow as the table containing its successful execution traces. More precisely, each entry of the TT describes a successful execution trace, which consists of a set of triples of the form \(\langle\text{Preconditions}, \text{Needed Inputs}, \text{Generated Outputs}\rangle\), where Preconditions represents the set of data and control constraints that must be satisfied to be able to successfully execute the workflow, in that execution trace. Needed Inputs and Generated Outputs are the set of inputs requested and outputs generated, respectively, by the tasks executed in the respective trace.

The process of generating the TT consists of looking in the RG (or MRT) of the workflow for all paths (i.e., traces) \(p\) originating in the initial marking and ending in the final marking. The preconditions set for \(p\) is given by the set of all conditions (viz., places) in the markings of \(p\). The set of needed inputs is obtained by taking the inputs of all tasks labelling arcs of the path \(p\). Similarly, the set of generated outputs consists of the outputs of all tasks labelling arcs of the path \(p\).

The TTs for the workflows in our example are given in Table 3.1.

Note that if there are loops (that do not generate unbounded workflows) in the RG then each loop is considered at most once. Loops that generate an infinite state-space are to be tackled with the MRT exclusively. For the workflow in Figure 3.4 we consider only two successful traces, as given by the TT in Table 3.2.

\(T_1\) comes from considering the RG path \(C_i \rightarrow C_2 \rightarrow C_o\), while \(T_2\) come from the path \(C_i \rightarrow C_1 \rightarrow C_3 \rightarrow C_2 \rightarrow C_o\). Please note that, although we consider cycles, we do not take into account tasks executed more than once. This is due to the fact

\(^4\)Slightly adapted so as to cope with YAWL workflow nets instead of Petri nets.
3.4. DESCRIPTION OF THE AGGREGATION APPROACH

**Search Engine** \( \{T_{SE}\} \): \(<\{C_1, C_2, C_3\}, \{fName, os\}, \{URL\}>.\)

**File Downloader** \( \{T_{FD}\} \): \(<\{C_4, C_5\}, \{URL\}, \{binaryData\}>.\)

**File Server** \( \{T_{FS}, T_{FS}'\} \): \(<\{C_6, C_7, C_9, C_{11}\}, \{fileName\}, \{limitedBandwidth, URI\}>,\)
\(<\{C_6, C_8, C_{10}, C_{11}\}, \{fileName\}, \{limitedBandwidth, file\}>.\)

**Fetch Application** \( \{T_{FA}\} \): \(<\{C_{12}, C_{13}, C_{14}\}, \{dataFile\}, \{appName, platform\}>.\)

Table 3.1: TTs of the example workflows.

<table>
<thead>
<tr>
<th>Task</th>
<th>TT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Search Engine ( T_{SE} )</td>
<td>(&lt;{C_1, C_2, C_3}, {fName, os}, {URL}&gt;.)</td>
</tr>
<tr>
<td>File Downloader ( T_{FD} )</td>
<td>(&lt;{C_4, C_5}, {URL}, {binaryData}&gt;.)</td>
</tr>
</tbody>
</table>
| File Server \( T_{FS}, T_{FS}' \)| \(<\{C_6, C_7, C_9, C_{11}\}, \{fileName\}, \{limitedBandwidth, URI\}>,\)
\(<\{C_6, C_8, C_{10}, C_{11}\}, \{fileName\}, \{limitedBandwidth, file\}>.\)
| Fetch Application \( T_{FA} \)| \(<\{C_{12}, C_{13}, C_{14}\}, \{dataFile\}, \{appName, platform\}>.\)

![Figure 3.4: Workflow with cycle.](image)

**Workflow with cycle** \( \{T_1, T_2\} \): \(<\{C_1, C_2, C_o\}, \{a\}, \circ>,\)
\(<\{C_t, C_1, C_2, C_3, C_o\}, \{a\}, \circ>.\)

Table 3.2: TT for the workflow in Figure 3.4.

that we are interested in gathering the inputs needed (collectively) for the execution of a workflow trace, and for this purpose it suffices executing a task only once.

An entry of the TT is to be read as follows. For example, the \(<\{C_6, C_7, C_9, C_{11}\}, \{fileName\}, \{limitedBandwidth, URI\}> \) trace of the File Server workflow in our example states that, “provided the fileName input, one may obtain the limitedBandwidth and URI outputs, if all conditions in the preconditions set are met”.

We say that a task \( T \) in (the workflow of) a service \( S \) is executed in a trace \( t \) if some precondition of \( t \) is an output place (i.e., condition) for \( T \) in the workflow of \( S \). For instance, the preconditions set \( \{C_6, C_7, C_9, C_{11}\} \) corresponds to the set of executed tasks \( \{Get Filename, Locate URI, Cache File\} \).
In order to provide a more user-friendly answer to the query, we construct a logical expression from the set of preconditions of a trace. We achieve this by firstly assigning a logical expression to each place of the workflow, and then by computing the conjunction of all the conditions in the preconditions set of a trace. For instance, the above preconditions set \( \{C_6, C_7, C_9, C_{11}\} \) might be simply expressed as \( \text{"limitedBandwidth OR (NOT Cached(fileName))"}\). To do so, by exploiting the usability of the YAWL predicates to enable tasks [87], we enhance the expressiveness of YAWL conditions by assigning them a logical expression. This process is to be done automatically as indicated in the following.

For the input and output conditions of a workflow we consider an always \( \text{"true"} \) condition. Furthermore, output places of tasks having an EMPTY- or an AND-split get an always \( \text{"true"} \) condition (e.g., \( C_9 \)). In the case of a XOR-split, we consider an output condition to be \( \text{true} \) provided \( \text{either the YAWL predicate for the corresponding link is true as well as the other lower-order predicates are false, or the corresponding predicate is the default one and all other predicates of the respective tasks are false} \). For example, for \( C_7 \) we consider the following expression \( \text{"limitedBandwidth OR ((limitedBandwidth = default) AND (NOT Cached(fileName)))"} \), or simply \( \text{"limitedBandwidth OR (NOT Cached(fileName))"} \). Hence, a token is placed into \( C_7 \) if the file is not cached, regardless of the bandwidth conditions. Similarly, for \( C_8 \) we have \( \text{"(Cached(fileName) AND (NOT limitedBandwidth)) OR ((Cached(fileName) = default) AND (NOT limitedBandwidth))"} \), or simply \( \text{"(Cached(fileName) AND (NOT limitedBandwidth))"} \). Last but not least, for a task having an OR-split, we consider an output condition to be \( \text{true} \) if and only if \( \text{its corresponding predicate is true, or the respective predicate is the default one, and all other predicates of the considered tasks are false} \) [87].

### 3.4.3 Service Matching

This phase deals with finding successful execution traces of the advertised services that could collectively satisfy, either fully or partially, the successful traces of the client service.

Consider a registry \( \{S_1, \ldots, S_n\} \) of service contracts, and a client contract \( C \). Furthermore, consider a set of successful traces \( T^S = \{t_1, \ldots, t_n\} \), where each \( t_i \) is a successful trace of some advertised service \( S_i \), and a set of successful client traces \( T^C = \{u_1, \ldots, u_m\} \). We say that \( T^S \) matches \( T^C \) if and only if the set of inputs needed collectively by all traces in \( T^S \cup T^C \) is included in the set of outputs generated collectively by them.

Note that set-theoretic union and inclusion (over sets of data) are ontology-aware. For example, \( \{fName\} \cup \{fileName\} = \{fileName\} = \{fName\} \) due to the assumed exact match between the two ontology types (Section 3.1). The union operation considers the less general type. For example, although we have assumed an exact match between URL and URI, we consider that \( \{URL\} \cup \{URI\} = \{URL\} \) because
URI is more general than URL\(^5\). This allows us to establish correctly whether the set of needed inputs can be obtained from the set of generated outputs using the following rule. According to the OWL-S specification [71], an output \(O_i\) is compatible with an input \(I_j\) if and only if either there is an exact match between \(O_i\) and \(I_j\) (viz., \(O_i\) and \(I_j\) represent the same concept, or \(O_i\) is a direct sub-concept of \(I_j\)), or if there is a plug-in match between \(O_i\) and \(I_j\) (viz., \(O_i\) is a non-direct sub-concept of \(I_j\). We say that there is a subsumes match between \(O_i\) and \(I_j\) if \(O_i\) is a parent concept of \(I_j\). In this case we do not have a match, as the output is more generic than the input. For example, assume that in an ontology of vehicles the car concept has a sports car direct sub-concept, which further has a rally car direct concept. Assume also that car has another camper sub-concept. Then, there is an exact match between a sports car output and a car input, as well as a plug-in match between a rally car output and a car input. Furthermore, there is a subsumes match between a car output and a sports car input. Note that the latter example does not offer a compatible match since at run-time, the service generating cars as output may output e.g., a camper, which is not a valid sports car input. Such considerations are also used by the inclusion relation.

The matching algorithm firstly tries to find candidate sets of traces of the advertised services that satisfy all client traces. In case no such candidate set exists, the algorithm looks for candidate sets that (partially) satisfy the maximum number of client traces. Consider that we want to match successful traces \(\{u_1, \ldots, u_m\}\) of the client service. We obtain the candidate sets using a Matchmaker Graph (or MG for short) as follows. A node of the MG consists of two sets. The first is a set of needed inputs, while the second is a set of generated outputs. A directed edge in the MG is labelled by a successful execution trace of a service in the registry. It connects one source and one target node. The inputs set of the target node is obtained by taking the union between the inputs set of the source node and the needed inputs set of the respective trace. The generated outputs set is obtained analogously. A requisite of the considered trace is that it has to satisfy at least one previously unconsidered input of the needed inputs set of the source node.

The MG is built by first considering the node \(N\) having as inputs set the inputs needed collectively by \(\{u_1, \ldots, u_m\}\) and, dually, the outputs set is made of the outputs generated by these traces. Further nodes \(N_k\) are obtained by looking for successful execution traces \(t_k\) of services \(S_k\) in the registry that satisfy at least one input needed in \(N\). The process of building the MG continues by considering the nodes \(N_k\), and it finishes when all nodes in the MG have either been considered or are final. A final node has the property that the set of needed inputs is contained in the set of generated outputs.

The MG obtained for our example, by considering the (only) successful execution trace of the Fetch Application client service (i.e., \(T_{FA}\)), is depicted in Figure 3.5. It shows that there are three candidate sets for fully satisfying the client request: (a)

\(^5\)We assume that URL is a direct (viz., child) sub-concept of URI.
\{T_{FD}, T_{SE}\}, which corresponds to executing the File Downloader and the Search Engine services, (b) \{T_{FD}, T_{FS}^1\}, which corresponds to executing the File Downloader and the File Server services, and (c) \{T_{FS}^2\}, which corresponds to executing the File Server service only.

The service matching phase is also in charge of automatically generating a data-flow mapping among service traces in a candidate set and the client ones. In order to derive data-flow information linking tasks of (possibly) different workflows, one has to match requested inputs with offered outputs. We call a match a data-flow dependency and a set of them a data-flow mapping. We recall that an output \(O_i\) is compatible with an input \(I_j\) if and only if either \(O_i\) and \(I_j\) represent the same concept, or \(O_i\) is a sub-concept of \(I_j\). Consequently, we consider only these types of matches. Matching IO parameters is achieved in two ways.

On the one hand, we employ a one-to-one matching between parameters of the tasks executed in the service traces previously mentioned. We recall that a task \(T\) in the workflow of a service \(S\) is executed in a trace \(t\) if some precondition of \(t\) is an output place (i.e., condition) for \(T\) in the workflow of \(S\).

On the other hand, further matches can be obtained using sets of equivalent parameter types given by the client. Such a set \(\{pType_1, \ldots, pType_x\}\) states that parameters of the \(pType_i\) type can be matched exactly by \(pType_j\), where \(pType_{i,j}\) are values in possibly different parameter ontologies, for each \(i,j \in \{1, \ldots, x\}\). In this way we allow for cross-ontology mappings. For example, consider that os_type and platform_type are the types of the os input parameter of the File Info task of the Search Engine workflow, and of the platform output parameter of the Set Platform
3.4. DESCRIPTION OF THE AGGREGATION APPROACH

Candidate Set \( \cup \) Client Service:

Data-flow dependencies.

\{Search Engine, File Downloader, Fetch Application\}:

- File Info(fName) \( <> \) Set Name(appName);
- File Info(os) \( <> \) Set Platform(platform);
- Download(URI) \( <> \) Download URL(URL);
- Get File(dataFile) \( <> \) Download(binaryData).

Table 3.3: Data-flow mapping for \{Search Engine, File Downloader, Fetch Application\}.

\{File Server, File Downloader, Fetch Application\}:

- Get Filename(fileName) \( <> \) Set Name(appName);
- Download(URI) \( <> \) Locate URI(URI);
- Get File(dataFile) \( <> \) Download(binaryData).

Table 3.4: Data-flow mapping for \{File Server, File Downloader, Fetch Application\}.

\{File Server, Fetch Application\}:

- Get Filename(fileName) \( <> \) Set Name(appName);
- Get File(dataFile) \( <> \) Download(binaryData).

Table 3.5: Data-flow mapping for \{File Server, Fetch Application\}.

task of the Fetch Application service, respectively. If we assume that the two types are defined in two distinct parameter ontologies, and that the client provides the set \{os\_type, platform\_type\} of equivalent ontology values, then we get an exact match between the two parameters.

We write a data-flow dependency between an input \( I \) of task \( P \) and an output \( O \) of task \( Q \) as \( "P(I) <> Q(O)" \). For simplicity, we assume here that all (task, parameter) name pairs are distinct. It is important to note that, for flexibility reasons, the client should be allowed to modify, cancel or add dependencies in the mapping. However, note that a data-flow mapping linking workflow tasks of all services in the registry can be done off-line. In this case, this phase has to match only the client inputs and outputs with the ones in the mapping done on the registry. The mappings generated for the three candidate sets and the client service are given in Table 3.3, 3.4, and 3.5, respectively.

The following two phases deal with generating the contract of the aggregated service and, respectively, its validation. For each candidate set, we have to compute the aggregation between the client contract and the contracts corresponding to each trace in the candidate set, and then to validate the aggregate. For example, for the candidate set \( \{T_{FD}, T_{FS}\} \) we have to aggregate the contract of the client Fetch
Application service with the the contracts of the File Downloader and File Server services.

3.4.4 Core Aggregation and Contract Generation

It is important to note that, for each candidate set of services, the methodology employs this phase of the aggregation for the generation of two composite contracts. On the one hand, it aggregates the services in the candidate set together with the service $C$ representing the client request into a composite contract $CS$. On the other hand, it aggregates only the services in the candidate set into a composite $S$. The reason for doing so is that, provided $SC$ does not (fully) satisfy the client request (see Subsection 3.4.5), the methodology attempts to generate an adapter $A$, which lets $S$ and $C$ to interact successfully. In other words, the adaptation process (see Chapter 4) concerns the generation of an adapter $A$ such that the composition of $A$ with $S$ and $C$ is lock-free.

The Core Aggregation and Contract Generation phase inputs a set of contracts to be aggregated and a data-flow mapping linking parameters of (possibly) different services, and it automatically generates the contract of the aggregated service in four steps:

1. **Task Expansion.** The first step expands all tasks with explicit control- and data-flow task constructs, also called Input/Output Control/Data enabler dummy tasks (or $ICs/IDs/OCs/ODs$ for short).

2. **Control-Flow Analysis.** The second step translates the initial flow dependencies of each workflow in terms of the newly added $IC$ and $OC$ dummies.

3. **Data-Flow Analysis.** The third step relates $IDs$ and $ODs$ of tasks belonging to (possibly) different workflows by taking into account the data-flow mapping.

4. **Contract Optimisation.** The fourth and final step clears the aggregated contract of redundant dummies and control constructs. The four steps are detailed hereafter.

### Task Expansion

The Task Expansion starts by considering the empty (aggregated) workflow $A$. Then, for each (atomic or composite) task $T$ of each workflow $W$, it applies the following algorithm:

1. Add to $A$ a copy of $T$, and call it $T^*$,

2. If $T$ has at least one input, then:
   
   (a) Set the join of $T^*$ to AND,
3.4. DESCRIPTION OF THE AGGREGATION APPROACH

(b) If the join of $T$ is not EMPTY or AND, add to $A$ an $IC$ that inherits the join of $T$, and call it $IC_T$. Then, add to $A$ a dependency link from $IC_T$ to $T^*$.

c) Add to $A$ an $ID$ that is in charge of gathering all inputs needed for the execution of $T$, and call it $ID_T$. If $T$ has more than one input, set the join of $ID_T$ to AND. Otherwise set it to EMPTY.

3. If $T$ has at least one output, then:

(a) Set the split of $T^*$ to AND,

(b) If the split of $T$ is not AND or EMPTY, add to $A$ an $OC$ that inherits the initial split of $T$, and call it $OC_T$,

(c) Add to $A$ an $OD$ that “offers” all outputs of $T$ to other tasks, and call it $OD_T$. Set the split of $OD_T$ to AND.

With the exception of $T^*$, all previously introduced tasks lack IOs and have void ontological values. Their purpose is to explicitly separate the control- and data-flow logic of $T$. From a flow point of view, $IC_T$ and $ID_T$ are linked as inputs of $T^*$ while $OC_T$ and $OD_T$ are linked to it as outputs.

Figure 3.6 describes the task expansion step applied to the Get Filename task of the File Server workflow. Get Filename* employs AND-join and split constructs as, on the one hand, Get Filename* can be executed only if it is enabled from the control-flow point of view (as we will see later) and if $ID_{Get Filename}$ has finished its execution and, on the other hand, both $OC_{Get Filename}$ and $OD_{Get Filename}$ are to be executed after Get Filename* terminates. One may also note the split of $OC_{Get Filename}$ that is the initial XOR-split of Get Filename. From a data-flow point of view, the EMPTY-join of $ID_{Get Filename}$ indicates that the fileName input of Get Filename must be available in order for it to execute. Dually, the AND-split of $OD_{Get Filename}$ specifies that, after Get Filename finishes executing, its output limitedBandwidth will be available to all tasks requesting it as input.

Once all tasks have been expanded, two more tasks are introduced. They are $IC_A$ and $OC_A$ corresponding to the input and the output control enabler dummies of $A$. $IC_A$ has an AND split in order to activate the $IC$s of all the workflows to be aggregated. Dually, $OC_A$ has an AND join in order to wait for the $OC$s of all the workflows to finish their execution. That is, if a task $T$ of a workflow $W$ was connected to the input/output condition of $W$, then the input/output control dummy of its expansion, $IC_T/OC_T$, has to be connected correspondingly to $IC_A/OC_A$. Furthermore, the input condition of $A$ has to be connected as input of $IC_A$, and $OC_A$ as input of the output condition of $A$.

All YAWL conditions in the initial workflows, with the exception of their input and output conditions, are copied without modifications into the aggregated workflow $A$. Note further that the Task Expansion step works the same for atomic and composite tasks, as well as for tasks with single or multiple instances. This is
because ICs and IDs represent necessary (control- and data-flow) “prerequisites” for the execution of tasks, and OCs and ODs represent the (control- and data-flow) “effects” of executing the tasks. Such prerequisites and effects do not vary with the type of the task. Furthermore, if \( T \) is a composite task, then only \( T \) is expanded, and not the tasks it contains. This is because YAWL does not allow links to cross the boundaries of composite tasks. In other words, links cannot have the source inside a composite task and the target outside it, or vice versa.

Task Expansion copes with cancellation sets (after all tasks have been expanded) as follows. For each task \( T \) (expanded into a set \( \{T^*, \text{IC}_T, \text{ID}_T, \text{OC}_T, \text{OD}_T\} \)) that is contained in the cancellation set of another task \( S \) (expanded into a set \( \{S^*, \text{IC}_S, \text{ID}_S, \text{OC}_S, \text{OD}_S\} \)), it adds all tasks of \( \{T^*, \text{IC}_T, \text{ID}_T, \text{OC}_T, \text{OD}_T\} \) to the cancellation set of \( S^* \). Furthermore, if a condition \( C \) belongs to the cancellation set of a task \( T \), then \( C \) will be contained in the cancellation set of \( T^* \) as well.

Control-Flow Analysis

During this step, the control-flow dependencies of each workflow \( W \) are specified in terms of the newly added ICs and OCs, as well as of IC\(_A\) and OC\(_A\).

Hence, for each workflow \( W \), and for each task \( T \) connected as input of another task \( S \) into \( W \), add to \( A \) a link that points from OC\(_T\) to IC\(_S\). Furthermore, if \( T \) was connected as input of a condition \( C \), then add a link that points from OC\(_T\) to \( C \). Analogously, if \( S \) was connected as output of a condition \( C \), then add another link from \( C \) to IC\(_S\). Note that, if \( T \) was not expanded with an OC\(_T\) dummy, then the source of the link will be \( T^* \) instead. Dually, if \( S \) was not expanded with an IC\(_S\) dummy, then the target of the link will be \( S^* \).

The result of applying this step on the File Server workflow may be seen in Figure 3.7.

For example, the initial control-flow link between Get Filename and Locate URI has been translated into a link between OC\(_{\text{Get Filename}}\) and Locate URI. Moreover, one should note that Get Filename and Cache File are now connected to IC\(_A\) and OC\(_A\), respectively. That is, IC\(_A\) enables (from the control-flow point of view) Get Filename for execution. Dually, the execution of Cache File is interpreted as the termination of the File Server service.
3.4. DESCRIPTION OF THE AGGREGATION APPROACH

The control-flow analysis for the Search Engine, File Downloader, and Fetch Application services are shown in Figure 3.8, 3.9, and 3.10, respectively.

**Data-Flow Analysis**

From a data-flow point of view, a prerequisite for executing a task $T$ is to have all its inputs available. The data-flow mapping obtained during the Service Matching phase can be expressed in terms of execution constraints between $IDs$ and $ODs$ as follows.

A data-flow mapping can be simply expressed as a set of pairs $((W, T, i), (Z, S, o))$, where $W$ and $Z$ are two workflows, $T$ and $S$ are, respectively, two of their tasks, and
i is an input of \( T \), and \( o \) is an output of \( S \). The purpose of this step is to express these mappings in terms of \( ID \) and \( OD \) dummies, as follows.

For each triple \((W, T, i)\) consider the set \( M \) of pairs \(((W, T, i), (Z, S, o))\) in the mapping. If \( M \) is void, choose another triple \((W, T, i)\). Otherwise, if \( M \) contains one element only, add to \( A \) a link from \( OD\_S \) to \( ID\_T \). Otherwise, (if \( M \) contains more than one element):

1. Add to \( A \) a dummy task \( T\_i \) with no IOs and with a void ontological value, but having a XOR-join and an EMPTY-split. This is due to the fact that a value for \( i \) may be obtained by executing different tasks \( S \), yet only one value is needed. Furthermore, add to \( A \) a link from \( T\_i \) to \( ID\_T \). For simplicity we assume that all \( T\_i \) names are distinct.

2. For each pair \(((W, T, i), (Z, S, o))\) in \( M \), add to \( A \) a link from \( OD\_S \) to \( T\_i \).

Figure 3.11 illustrates the data-flow mappings of our example expressed in terms of links between \( IDs \) and \( ODs \).

At the end of this phase one obtains a “rough” workflow of the aggregated service. The YAWL workflows of the three aggregated services of our example are depicted...
3.4. DESCRIPTION OF THE AGGREGATION APPROACH

Figure 3.12: AggS1: Workflow obtained by aggregating the Search Engine, the File Downloader, and the Fetch Application workflows.

Figure 3.13: AggS2: Workflow obtained by aggregating the File Server, the File Downloader, and the Fetch Application workflows.

in Figure 3.12, 3.13, and 3.14, respectively. As previously mentioned, the signature and the ontology information of the aggregated are to be obtained from the union of the signatures and of the ontology descriptions, respectively, of the services to be aggregated.

Contract Optimisation

The three steps before constructed a rough contract of the aggregated service. This last step is in charge of (repeatedly) removing from the aggregated contract
redundant dummies and join/split control constructs introduced previously. One obtains at the end of this step the “optimised” service contract $A^6$. We briefly describe hereafter the two redundancy elimination criteria.

**Dummy absorption.** Assume a dummy (i.e., control- or data-flow enabler, or $T_i$ dummy added during the data-flow analysis) $iD$ connected as input of task $T$ such that the pair $<\text{join}_{iD}, \text{join}_T>$ matches the following – $\{<\text{EMPTY}, \text{EMPTY}>, <\text{EMPTY}, \alpha>, <\alpha, \alpha>\}$ –, where $\alpha \in \{\text{AND}, \text{XOR}, \text{OR}\}$. Then, the dummy $iD$ is “absorbed” into $T$, which remains unchanged. Absorption means that $iD$ is removed from $A$, and all tasks that were targeting $iD$ (if any), now have to target $T$. If $<\text{join}_{iD}, \text{join}_T>$ matches $<\alpha, \text{EMPTY}>$, then $iD$ is absorbed into $T$ with the observation that $T$ inherits the join of $iD$ (i.e., $\text{join}_T := \text{join}_{iD}$). The scenario is dual for absorbing output dummies. This criteria can be applied for clearing all $ID$s and $OD$s of the aggregated services depicted in Figures 3.12–3.14.

**Join/Split elimination.** A $\text{join}_T \neq \text{EMPTY}$ has to be set to EMPTY provided $T$ has only one incoming link. The dual (i.e., the “reset” of $\text{split}_T$ given $T$ has at most one outgoing link) is resolved in a similar way. For example, the $\text{SetPlatform}^*$, $\text{GetFilename}^*$, and $\text{SendFile}^*$ tasks of the aggregated service in Figure 3.13 get their AND-splits reset to EMPTY ones after previously absorbing their $OD$s.

The optimised YAWL workflows (augmented with explicit conditions) of the three aggregated services of our example, as well as their RGs, are depicted in Figure 3.15, 3.16, and 3.17, respectively.

---

$^6$Please note that the optimisation process here does not concern the generation of the “optimal aggregated workflow”, which may be a topic for future work. It simply clears the rough workflow of redundant constructs.
3.4.5 Contract Validation

For each aggregated contract $A$ previously obtained by composing a set $\{S_1, \ldots, S_n\}$ of advertised services with the client service $C$, we have to verify whether the successful traces of $A$ satisfy the previously matched successful traces of $C$. We achieve this by generating the TT of the aggregate $A$ and by verifying its compatibility with the TT of $C$. Informally, we have to check for each previously matched successful trace $u$ of the client whether all tasks executed in $u$ are executed in at least one successful trace $t$ of the aggregate. We recall that a task $T$ of a service $S$ is executed in a trace $t$ if some precondition of $t$ is an output place (i.e., condition) for $T$ in the workflow of $S$. More precisely, for each entry $u$ corresponding to a matched successful trace of the client service $C$ we have to verify whether there exists at least one entry $t$ corresponding to a successful trace of the aggregated service $A$, such that all tasks executed by $u$ are executed by $t$ as well.

We say that the client service is fully satisfied if all its successful traces are satisfied. Similarly, the client is partially satisfied if some yet not all of its traces are satisfied. If none of its traces are satisfied we say that the client is not satisfied. For the former two cases we say that the aggregation is successful, while for the latter case we call it a failure. For each satisfied trace $u$ of the client service $C$, our methodology replies with a concrete answer:
**YES:** the aggregation of the services in A fulfils the trace \( u \) of the C client service. – if \( u \) is not constrained by any preconditions set,

**MAYBE:** the aggregation of the services in A may fulfil the requested trace \( u \) of the C client service [with condition \( \text{Precond}_1 \land \ldots \land \text{Precond}_N \)] – if \( u \) is conditioned by at least one precondition set \( \text{PS}_k \), and the logical expression \( \text{Precond}_1 \land \ldots \land \text{Precond}_N \) is obtained by the conjunction of the conditions in \( \text{PS}_k \).

Please note that the condition constraining the fulfilment of the request is displayed if and only if its logical expression form has as operands only variables defined by the client request. In this way we avoid outputting a result which may not be understandable by the client.

The final result of the aggregation process is a list of successful service aggregations that fully/partially satisfy the client service. The output list is ordered by the number of unconstrained satisfied client traces, that is, the number of **YES** answers. One should note that the output list could be ordered further with respect to client’s preferences such as the number of conditions constraining the fulfilment of the request, or the number of services involved in the aggregation, and...
3.4. DESCRIPTION OF THE AGGREGATION APPROACH

Figure 3.17: Optimised workflow and RG for the AggS₃ service.

Table 3.6: TT of the aggregated service AggS₁ (Figure 3.15).

Table 3.7: TT of the aggregated service AggS₂ (Figure 3.16).

so on. If no client traces can be satisfied, the algorithm replies with the following answer: There are no services in the registry that can be successfully aggregated to fully/partially satisfy the request.

Tables 3.6–3.8 present the TTs of the three aggregated services of our example. By inspecting the only successful execution trace of the client service (note the Fetch Application TT entry in Table 3.1) we get that the set of tasks executed for satisfying the client request is \( Tasks_C = \{Set \text{ Name}, Set \text{ Platform}, Get \text{ File}\} \). Similarly, by inspecting the successful execution traces of the three aggregated services (note the three TTs Table 3.6, Table 3.7, and respectively Table 3.8), we obtain the following
sets of tasks that have to be executed for the successful execution of the three aggregated services:

- \( \text{Tasks}_{\text{AggS}_1} = \{ \text{IC}_A, \text{Set Name}^*, \text{Set Platform}^*, \text{Get File}^*, \text{File Info}^*, \text{Download URL}^*, \text{Download}^*, \text{OC}_A \} \),

- \( \text{Tasks}_{\text{AggS}_2} = \{ \text{IC}_A, \text{Set Name}^*, \text{Set Platform}^*, \text{Get File}^*, \text{Get Filename}^*, \text{OC}_{\text{Get Filename}}, \text{Locate URI}^*, \text{Cache File}^*, \text{Download}^*, \text{OC}_A \} \), and

- \( \text{Tasks}_{\text{AggS}_3} = \{ \text{IC}_A, \text{Set Name}^*, \text{Set Platform}^*, \text{Get File}^*, \text{Get Filename}^*, \text{OC}_{\text{Get Filename}}, \text{Send File}^*, \text{Cache File}^*, \text{OC}_A \} \).

One should note that all three candidate aggregates fully satisfy the client request as the set \( \text{Tasks}_C \) is included in \( \text{Tasks}_{\text{AggS}_1} \), \( \text{Tasks}_{\text{AggS}_2} \), and \( \text{Tasks}_{\text{AggS}_3} \), respectively. Moreover, for the only successful trace (call it \( T_{FA} \)) of the Fetch Application client service, the aggregation outputs the following ordered list:

1. **YES**: the aggregation of the services in \{Search Engine, File Downloader\} fulfils the trace \( T_{FA} \) of the Fetch Application client service.

2. **MAYBE**: the aggregation of the services in \{File Server, File Downloader\} may fulfil the trace \( T_{FA} \) of the Fetch Application client service.

   The logical expression "\( \text{limitedBandwidth OR (NOT Cached(fileName))} \)" constraining the fulfilment of the request is obtained by computing the conjunction of all conditions in the preconditions set of the (unique) TT entry of the aggregated service \( \text{AggS}_2 \) (note Table 3.7). We do not output it as it refers the variable \( \text{limitedBandwidth} \) as well as the \( \text{Cached}(...) \) method unknown to the client.

3. **MAYBE**: the aggregation of the services in \{File Server\} may fulfil the trace \( T_{FA} \) of the Fetch Application client service. The logical expression "\( \text{Cached(fileName) AND (NOT limitedBandwidth)} \)" is obtained by computing the conjunction of all conditions in the preconditions set of the (unique) TT entry of the aggregated service \( \text{AggS}_3 \) (note Table 3.8). Similarly to the previous case, we do not output this condition.
3.4.6 Informal Proof of Correctness

In the following we informally prove the correctness of the aggregation process. In particular, we assume that the aggregation generates a composite workflow $S_{\text{Agg}}$ obtained from the composition of $\{S_1, \ldots, S_n, Q\}$, where $S_k$ are workflows of contracts in a registry, and $Q$ is the workflow of the query contract. Since $S_{\text{Agg}}$ is an output of the methodology, it means that there exists at least one trace $T_{S_{\text{Agg}}}$ of $S_{\text{Agg}}$ that fulfills at least one trace $T_Q$ of $Q$. In short, this means that the execution of the tasks of $T_{S_{\text{Agg}}}$ provides all necessary inputs for the execution of the tasks of $T_Q$, and that the inputs needed for the execution of the tasks $t_k$ of $T_{S_{\text{Agg}}}$ are obtained from the outputs generated by the execution of tasks $t_x$ in $T_{S_{\text{Agg}}}$, where $x < k$. Hence, we show that any task $t_k$ belonging to the trace $T_{S_{\text{Agg}}}$ has its inputs satisfied due to the execution of other tasks $t_x$ of $T_{S_{\text{Agg}}}$, where $x < k$. Note that this suffices since the tasks of $T_Q$ are also tasks of $T_{S_{\text{Agg}}}$ (due to the Contract Validation phase).

Proof. The proof is immediate, by construction. Every input parameter $i$ of $T_k$ is reflected in $S_{\text{Agg}}$ by data-flow dependencies emerging at tasks $T_x$ and reaching $T_k$ (possibly through intermediary data-flow dummy enabler tasks), where each task $T_x$ has an output $o$ that matches $i$. This comes from the Functional Analysis phase (that makes sure that $S_{\text{Agg}}$ is a “closed workflow” in terms of data-flow) and from the Data-Flow Analysis step (that expresses the match between $i$ and $o$ as dependencies between the data-flow dummy enablers of the two tasks, and hence indirectly between $T_x$ and $T_k$). $T_k$ is a task of $T_{S_{\text{Agg}}}$, hence in the workflow simulation employed by the reachability analysis for the generation of $T_{S_{\text{Agg}}}$, the task $T_k$ receives at least one token on a data-flow dependency link from a task $T_x$. Consequently, we get that at least one task $T_x$ was previously executed in the respective simulation of $S_{\text{Agg}}$, hence $T_x$ is a task of the trace $T_{S_{\text{Agg}}}$ and $x < k$. 

3.5 Examples

In this Section we thoroughly describe a few aggregation scenarios in order to illustrate how the aggregation copes with various YAWL constructs.\(^8\)

3.5.1 First Example

Figure 3.18 presents a simple example, in which all workflows consist only of atomic tasks. The BookStore workflow describes the protocol of a service that sells books. When executed, the token placed in its input condition enables for execution the GetCatalogue task, which outputs a catalogue value. The token placed in the following deferred choice \(^7\) enables for execution several tasks. If the client chooses to

\(^7\)The correctness of the MRT generation is given in [96].

\(^8\)The interested reader can download the examples described in this Section from http://www.di.unipi.it/~popescu/Sator_Examples.zip.
Figure 3.18: Example illustrating the workflows of three (interacting) Web services.
execute the *GetBookPrice* task, then the workflow inputs the *title* of a book (from the client) and it outputs its *price* (to the client). Similarly, the *Add2ShoppingCart* task inputs the title of the book the client wishes to buy, while *ResetShoppingCart* removes all items previously added to the cart.

If the client does a *Checkout*, then the workflow will output a *totalPrice*, which is the cost of the books in the cart. Next, a token is placed in the deferred choice following the *Checkout* task. Now, the client has the possibility to invoke, *either* one of the *GetBookPrice*, *Add2ShoppingCart*, *ResetShoppingCart*, *Checkout*, or *Exit* tasks, *or* the *ConfirmOrder* task. Note that the execution of any of the former five tasks leads to the removal of the *ConfirmOrder* tasks from the list of tasks that can be executed by the client. This is due to the fact that their execution consumes the token in the input condition of the *ConfirmOrder* task.

The *ConfirmOrder* task, whose execution has to immediately follow the execution of the *Checkout* task, inputs the credit card information (*ccDetails*), as well as the client’s address used for delivery (*deliveryInfo*), and it outputs the *paymentDetails*, which (as we shall see later) are to be used by the *Bank* service to verify the validity of the transaction. The order confirmation is followed by the execution of the *SetOrderStatus* task, which inputs the response of the *Bank* service (*orderStatus*), and it outputs a *receipt* to the client. If the *Bank* approved the transaction, the execution continues with the *Exit* task which logically marks the end of a buying session. Otherwise, a token is placed into the first deferred choice. Note that the client has also the possibility of terminating the buying session by executing the *Exit* task at any moment after the execution of the *GetCatalogue* task, but while it is waiting for a *receipt* from the *BookStore*.

The second workflow in the example describes a *Bank* service that can be accessed, for example, by the *BookStore* in order to validate the credit of a book-buayer. The execution of the *Bank* workflow starts with the execution of the *SetPaymentDetails* task, which inputs the *paymentDetails* as well as the *total* price to be paid by the person indicated in the *paymentDetails*. The execution continues with the *ValidateCC* task, which verifies the credit card information (e.g., the credit card number and validity period), and it outputs a *flag* that is used internally by the *Bank* workflow to determine the control-flow. A “KO” value of the *flag* indicates that the supplied credit card information is *not* valid, and the execution continues with the *GetResponse* task, which outputs to the client (i.e., invoker) of the *Bank* service a corresponding *orderStatus* response. Otherwise, an “OK” value of the *flag* leads to the execution of the *VerifyFunds* task, which checks, for example, whether the book-buayer can afford paying the books.

Please note that, in order to ease the presentation, we did not represent all the task inputs and outputs (IOs), such as the *total* output of *SetPaymentDetails*, which has to be (at a later moment) inputted by the *VerifyFunds* task as well. (All such YAWL mapping details such as passing values among internal tasks of a workflow have been left out intentionally.) Finally, the execution of the *VerifyFunds* task leads to the termination of the workflow due the execution of the *GetResponse* task.
The third workflow depicts a simple Client service that attempts to buy a book from an e.g., BookStore service. At the start of the workflow, the invoker of the Client service executes the GetBook task, which outputs the title of the desired book (bookTitle). Next, the SetPrice task inputs the price of the respective book, and depending on its value, the execution continues with one of the following two scenarios. On the one hand, should the book price not exceed a certain amount of money (e.g., 49.99 euros), the invoker has to execute in any order she wishes the Payment and the GetDeliveryInfo tasks. The former outputs the invoker’s credit card details (ccDetails), while the latter outputs the address where the book is to be delivered (deliveryInfo). In this scenario, the workflow continues with the execution of the SetReceipt task, which waits for a receipt for the book being bought, and then with the Exit task. On the other hand, if the book price is higher than the predefined amount, the workflow finishes with the execution of the Exit task.

Assume a book-buyer is in possession of a Client service that she wants to use for buying a book. However, in order to successfully complete such action, the Client service has to obtain the price of the book and, assuming that it costs less than 50 euros, it has to receive also a receipt for the transaction. Values for these inputs are to be given by outputs of another service(s), such as the BookStore. For example, the price output of its GetBookPrice task can be used as an input for the SetPrice task of the Client service. Furthermore, the receipt outputted by the SetOrderStatus task of BookStore may serve as input for the SetReceipt task of Client. Still, note that, in order to successfully execute, the BookStore service is constrained by obtaining values for the inputs of its ConfirmOrder and SetOrderStatus tasks. While the two inputs of the former are to be provided by the Client service, the input of the latter could be obtained from the GetResponse task of the Bank service. However, in order to output an orderStatus, the Bank service first needs values for the two input parameters of its SetPaymentDetails task, both of which can be obtained from the BookStore.

It is worth noting that, given the Client service, there are at least two possible scenarios for selecting the BookStore and the Bank services from a registry of service (contracts) advertisements. On the one hand, one can manually browse a UDDI registry of service contracts, while on the other hand, one can use an ontology-aware matching algorithm for such purpose. We recall that we argue for services described by contracts that contain ontology information about the service input and output parameters. In Section 3.4 we showed how service execution traces can be derived from service contracts and then matched in order to locate services that collectively can satisfy a query represented as another service. For our example, the Client service can be used as a query that leads to the selection of the BookStore and the Bank services.

For simplicity we assume only exact matches [73] among the IOs of the three services. The IO matches are illustrated in Figure 3.19. For example, the match between the price input of Client and the totalPrice output of BookStore (second row
### Figure 3.19: IO matches of the three services.

In the IO matches table) leads to considering the BookStore service as a candidate for (collectively) satisfying (together with other matched services) the Client service. Furthermore, the match between the orderStatus input of BookStore and the orderStatus output of Bank leads to adding the Bank service to the candidates list. The candidate set containing the BookStore and the Bank service is a valid candidate set because the set of inputs needed collectively by the two services, together with the Client one is contained in the set of outputs generated by them.

Now, if we assume that e.g., the bookTitle and the title ontology concepts do not belong to the same ontology, the matching algorithm would not be able to automatically match them. However, the matchmaker described in Subsection 3.4.3 is able to match the two concepts if the client (of the aggregation and adaptation methodology) provides the set \{bookTitle, title\} as a set of equivalent ontology concepts. We recall that we use such sets of equivalent concepts so as to cope with cross-ontology mappings. It is important to note further that the client is allowed to modify, append, and/or remove matches from the matches table. For example,
the match between the price input of the SetPrice task of the Client service and the price output of the GetBookPrice task of the BookStore service should be removed by the client from the table of IO matches, as the totalPrice of the Checkout task of the BookStore service actually reflects the amount of money that the client has to pay for the book, since we assume a constant delivery cost which is included in the totalPrice.

Hereafter we describe the aggregation of the workflows given in Figure 3.18 assuming the data-flow mapping given in Figure 3.19 from which we have removed the GetBookPrice(...)::price from the second row, the BookStore’s column.

**Task Expansion**

We recall that the Task Expansion step serves to explicitly split the control- from the data-flow dependencies. The Task Expansion step applied to all the tasks of the three workflows to be aggregated yields the tasks in Figure 3.20.

For example, the GetCatalogue* has only an OD as it does not have any inputs yet it does have an output. Furthermore, GetBookPrice* has both an ID and an OD as it has both inputs and outputs. The join of its ID is EMPTY as GetBookPrice has one input only. Another example is SetOrderStatus*, which in addition to an ID and an OD, it gets an OC, which inherits the original (XOR) split of SetOrderStatus.
The *Exit* tasks of both the *BookStore* and the *Client* workflows are not expanded as they do not have any inputs or outputs.

The expansion of e.g., the *SetOrderStatus* task is to be interpreted as follows: the *ID* serves for waiting for a value for the *orderStatus* input. The AND-join of *SetOrderStatus* is needed to enable *SetOrderStatus* only when both the control- and the data-flow constraints are met. In other words, *SetOrderStatus* can be executed only when it gets enabled from the control-flow point-of-view, and a value has been assigned to its input parameter. Dually, its AND-split serves to enable the *OC* and *OD* dummies. The *OC* logically marks the termination of *SetOrderStatus*, while the *OD* is used to “broadcast” (due to the AND-split) the value of its output parameter.

Note that the Task Expansion step does not apply to the two conditions of the *BookStore*. Furthermore, its unnamed dummy task is not expanded as it does not have any IOs.

**Control-Flow Analysis**

The Control-Flow Analysis step translates the initial control-flow dependencies of each workflow to be aggregated into control-flow dependencies among control-flow dummies of the expanded dummies. A task should be used instead of the dummies if the respective task was not expanded with control-flow dummies. By applying the Control-Flow Analysis on the three workflows of our example one gets the (partial) workflow in Figure 3.21.

**Data-Flow Analysis**

The Data-Flow Analysis step applied to our example translates the matches among the IOs of the services to be aggregated (see Figure 3.19) into dependencies among *IDs* and *ODs* of the corresponding expanded tasks. These dependencies are illustrated in Figure 3.22.

**Contract Optimisation**

The YAWL workflow of the aggregate one obtains for our example (at the end of the Data-Flow Analysis step) is given in Figure 3.23. We call it the “rough” workflow of the aggregate. Please note that the *ODs* of *ValidateCC* and *GetCatalogue* do not have outgoing links as there are no tasks whose inputs match their outputs.

For example, during the Contract Optimisation step, the dummy absorption removes both the *ID* and the *OD* of the *GetBookPrice* task as, on the one hand, the *ID* has an EMPTY-join while *GetBookPrice* has an AND-join, and on the other hand, both the *OD* and the *GetBookPrice* tasks have an AND-split. Then, the Join/Split elimination criterion resets the split of *GetBookPrice* to EMPTY as it has only one outgoing link.
CHAPTER 3. WEB SERVICE AGGREGATION

Figure 3.21: Applying the Control-Flow step on the example workflows.

Furthermore, we can absorb the OD of GetCatalogue*, the ID of Add2ShoppingCart*, the OD of Checkout*, the ID and OD of ConfirmOrder* and SetOrderStatus*, the ID of SetPaymentDetails*, the OD of ValidateCC*, the OD of GetResponse*, the OD of GetBook*, the ID of SetPrice*, the ODs of Payment* and GetDeliveryInfo*, as well as the ID of SetReceipt*. Finally, we can reset to EMPTY the AND-splits of GetCatalogue* and ValidateCC*.

At the end of the Contract Optimisation step one gets the workflow in Figure 3.24. The only dummy which remains in the final workflow of the aggregate is the OC of the SetOrderStatus* task. The explicit separation of the control- and the data-flow is necessary in this case as one cannot simply link the SetOrderStatus* task as input of the output place of GetCatalogue* task, of the Exit* task (of the BookStore service), and of the SetReceipt* task. While the execution of SetOrderStatus* outputs a token to only one of the former two, it always sends a token to the latter. (In other words, the OC cannot be absorbed into SetOrderStatus* as their splits are not compatible.)

The example presented so far contains atomic processes only. In the following we shall describe three more examples that show how the aggregation copes with:

- composite tasks,
3.5.2 Second Example

The second example we describe in this Section is presented in 3.25. As previously mentioned, the differences with respect to the first example are that here, the initial Bank workflow has been wrapped as a composite task, and finally, by adding a cancellation set to the new Client workflow.

The three examples are obtained from the first one, initially by wrapping the Bank service into a composite task, then by modifying the Client and BookStore workflows such that the BookStore includes a multiple-instance task, and finally, by adding a cancellation set to the new Client workflow.

Figure 3.22: Transformation of the IO matches into dependencies among IDs and ODs.
task in their application workflows, without having to link the internal tasks of the Bank2 workflow into their workflows.

Please note that in the following we shall partially explain the aggregation steps by showing the differences with respect to the previous example.

**Task Expansion**

We recall that the Task Expansion step serves to explicitly split the control-flow from the data-flow dependencies. The Task Expansion step applied to the Bank2 workflow gives the three tasks in Figure 3.26. We recall that only the tasks of the top-level workflow net [87] can be expanded. In other words, the Task Expansion step cannot be applied to the workflow net of the ExecutePayment composite task as links that might be added during the Data-Flow Analysis cannot cross the boundary of the composite task. Note in Figure 3.26 the AND-join of the ID, which is due to the fact that ExecutePayment has two inputs.

**Control-Flow Analysis**

The Control-Flow Analysis step translates the initial control-flow dependencies of each workflow to be aggregated into dependencies among control-flow dummies of
the expanded tasks. A task should be used instead of the dummies if the respective task was not expanded with control-flow dummies. The Control-Flow Analysis yields the (partial) workflow given in Figure 3.27. Note the simplification of the workflow due to the encapsulation of the Bank service logic as a composite task. The initial control-flow links between the ExecutePayment task and the input and output conditions of the Bank2 workflow are translated into links between the expanded ExecutePayment* task and the IC_A and the OC_A, respectively, dummies of the aggregate.

Data-Flow Analysis

The new matches among IOs of the three services in Figure 3.25 are presented in Figure 3.28. Observe that the Bank2 column refers only to the ExecutePayment task. Furthermore, the last row of the IO matches table of the first example (see Figure 3.19) is not included in the new table, as the Validate_CC task outputting the flag parameter is now “hidden” inside the ExecutePayment task.

Consequently, the Data-Flow Analysis now links the ODs of Checkout* and ConfirmOrder* with the ID of ExecutePayment*, and the OD of ExecutePayment* with the ID of SetOrderStatus*. The transformation of the IO matches into workflow dependencies linking IDs and ODs is given in Figure 3.29.
Figure 3.25: Another example for illustrating how the aggregation copes with composite tasks.
3.5. EXAMPLES

Figure 3.26: Task Expansion step applied to the Bank2 service.

Figure 3.27: Control-Flow Analysis of the example workflows in Figure 3.25.

Furthermore, the “rough” workflow of the new aggregated service, which one obtains at the end of this step, is given in Figure 3.30.

**Contract Optimisation**

This step is quite similar to the previous example. The Dummy Absorption and Join/Split Elimination criteria remove all the redundant IDs and ODS and reset to EMPTY the joins and splits with one input and output, respectively. Figure 3.31 shows the final YAWL workflow of the aggregate. Note that in order to execute the ExecutePayment* task one has to enable it first, both from the control- and the data-flow viewpoints. The former relates to executing the IC_A task, while the latter to executing the Checkout* and the ConfirmOrder* tasks.
### 3.5.3 Third Example

The third example is introduced in Figure 3.32. As previously mentioned, in this example we modify the Client and the BookStore services. They are called now Client2 and BookStore2, respectively.

The Client2 workflow starts with the execution of the ChooseBooks task, which inputs a catalogue of books and it outputs a list of books to be bought. Next, the CheckTotal task waits for the user to input a maximum price to be paid for these books (maxPrice), as well as it waits for a list of book prices from the BookStore2 service. The booksPrice output of CheckTotal stands for the total cost of the books (excluding delivery costs). Then, the control-flow is decided based on the booksPrice and on the maxPrice. On the one hand, if all books can be bought, Client2 first executes GetPaymentDetails, which outputs the delivery information and the card details, and then it executes the SetReceipt task, which inputs the receipt from the BookStore2 service. On the other hand, if booksPrice exceeds maxPrice, the execution of the workflow continues with a deferred choice. The invoker of the Client2 service has to decide whether to exit by executing the End task, or to remove some of the books from the selectedBooks list. RefineBookList inputs priceList so as
Figure 3.29: Transformation of the IO matches into dependencies among IDs and ODs.

to ease the job of the invoker by displaying the price of each book in the list. The execution continues next with the CheckTotal task.

The main difference between the (new) BookStore2 and the (old) BookStore workflows, is that BookStore2 has a multiple-instance task – GetBookPrices, which inputs a list of books and it outputs a list containing their prices. We assume that the number of instances of the GetBookPrices task is fixed and equal to the size of the bookList, same as the lower and the upper bounds of the number of instances created after the initiation of the task, and the threshold value that decides when the GetBookPrices task completes its execution. (For further in-depth information on multiple-instance tasks please see [87].) Hence, each book in the bookList leads to an instance of the GetBookPrices task, which outputs the book’s price. When all instances have finished their executions, the output of GetBookPrices is obtained by merging the individual book prices into the priceList. This behaviour is achieved by suitably mapping the IOs of the GetBookPrices task and of the workflow net of the BookStore2 service.
Figure 3.30: The rough workflow of the service obtained by aggregating the three services in Figure 3.25.

The second difference between the two workflows is that the Add2ShoppingCart task of BookStore2 inputs a list of books to be added into the shopping cart.

**TaskExpansion**

The Task Expansion step expands the tasks of the three workflows as shown in the previous examples. Consequently, we shall present here only the expansion of the multiple-instance task GetBookPrices of the BookStore2 workflow. As illustrated in Figure 3.33, GetBookPrices\(^*\) employs AND-join and -split constructs, as well as it is connected with an ID and an OD task.

Informally, the ID enables GetBookPrice\(^*\) from the data-flow point-of-view, that is, it waits for a value to be mapped to the bookList input parameter of GetBookPrice\(^*\), while the OD broadcasts its priceList output. Hence, from the Task Expansion viewpoint, a multiple-instance task (similarly to a composite task) looks exactly like a simple atomic task.
Figure 3.31: The final workflow of the service obtained by aggregating the three services in Figure 3.25.

### Control-Flow Analysis

This step builds (part of) the control-flow of the aggregate by translating the initial control-flow links among workflow tasks into links among ICs and OCs. Applying this step to the third example in Figure 3.32 yields the partial workflow in Figure 3.34.

### Data-Flow Analysis

Matching the IO parameters of the workflows in this example leads to the table in Figure 3.35. One may see that the resulting table is slightly more complicated with respect to the previous examples due to the increased number of matches. For example, the fifth row describes the fact that the `selectedBooks` input of `RefineBookList` matches similar outputs of `ChooseBooks` and `RefineBookList` of the same workflow (`Client2`), as well as the `bookList` outputs of the `GetBookPrices` and `Add2ShoppingCart` tasks of the `BookStore2` workflow.

In this example we shall assume that the client of the aggregation process removes only the matches between the `maxPrice` input and `booksPrice` output of the
Figure 3.32: Example for illustrating how the aggregation copes with multiple-instance tasks.

CheckTotal task (of the Client2 workflow) with the totalPrice output of the Checkout task (of the BookStore2 workflow), and with the total input of the ExecutePayment task (of the BankService2 workflow). In other words, the second row, first column of the table in Figure 3.35 is set to void. On the one hand, the removal of maxPrice is (mainly) motivated by the fact that it is an input of the Client2 service, whose value has to be provided by the invoker of the Client2 service, and not taken from the output of another service in the aggregation. On the other hand, the removal of booksPrice is due to the fact that it is an internal flag-variable used to decide the control-flow following the CheckTotal task.

Figure 3.33: Expanding the GetBookPrices task of the BookStore2 workflow.
3.5. EXAMPLES

The ontology matches in Figure 3.35 (after removing the unwanted matches) relate in terms of dependencies among ID and OD dummies as shown in Figure 3.36. Please note the dummies necessary when an input matches several outputs. This is the case for the bookList inputs of GetBookPrices and Add2ShoppingCart, as well as for the selectedBooks input of RefineBookList. Each such input dummy has a XOR-join as one (output) value only is enough for mapping the respective (input) parameter. For example, a bookList input for GetBookPrices can be obtained either from the output of ChooseBooks, or from the output of RefineBookList.

It is sometimes the case that some of the data dependencies are redundant. This is the case of the (data-flow) loop created around RefineBookList due to the match between its selectedBooks input and its selectedBooks output. Usually, avoiding the generation of such loops is the task of the aggregation client. She can either check the data-flow mapping (viz., the IO matches table) “by hand”, or she can use a reachability analysis (as described in Section 3.4) e.g., for the detection of (dead-)locks in the aggregated workflow. Should a (dead-)lock exist, she can (manually) remove the troublesome match(es) from the data-flow mapping, and then redo the (automated) core aggregation process.

The rough aggregated workflow reflecting both control- and data-flow dependen-
### Contract Optimisation

After removing redundant dummies as well as redundant joins and splits from the rough workflow of the aggregate, one obtains the workflow in Figure 3.38. Note that the aggregation removes the IDs of `GetBookPrices`, `Add2ShoppingCart`, and `RefineBookList`, yet not their input dummies added during the Data-Flow Analysis phase (i.e., `GetBookPrices_bookList`, `Add2ShoppingCart_bookList`, and `RefineBookList`).

---

**Figure 3.35:** The IO matches table among parameters of the three services in Figure 3.32.

<table>
<thead>
<tr>
<th>Client2</th>
<th>BookStore2</th>
<th>Bank2</th>
</tr>
</thead>
<tbody>
<tr>
<td>ChooseBooks(catalogue)...</td>
<td>GetCatalogue():catalogue</td>
<td>-</td>
</tr>
<tr>
<td>CheckTotal(maxPrice)...</td>
<td>Checkout():totalPrice</td>
<td>ExecutePayment(total)...</td>
</tr>
<tr>
<td>CheckTotal(booksPrice)...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CheckTotal(priceList)...</td>
<td>GetBookPrices():priceList</td>
<td>-</td>
</tr>
<tr>
<td>RefineBookList(priceList)...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SetReceipt(receipt)</td>
<td>SetOrderStatus():receipt</td>
<td>-</td>
</tr>
<tr>
<td>RefineBookList(selectedBooks)...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ChooseBooks():selectedBooks</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RefineBookList():selectedBooks</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GetPaymentDetails():ccDetails</td>
<td>ConfirmOrder(ccDetails)...</td>
<td>-</td>
</tr>
<tr>
<td>GetPaymentDetails():deliveryInfo</td>
<td>ConfirmOrder(deliveryInfo)...</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>ConfirmOrder():paymentDetails</td>
<td>ExecutePayment(paymentDetails)...</td>
</tr>
<tr>
<td>-</td>
<td>SetOrderStatus(orderStatus)...</td>
<td>ExecutePayment():orderStatus</td>
</tr>
</tbody>
</table>

Scenarios among the participant services is given in Figure 3.37. Please note that the dummy task joining in input the ODs of `ChooseBooks` and `RefineBookList` has not been produced by the aggregation process. We use it here just for simplifying slightly the graphical representation of the control-flow of the aggregated service.
Figure 3.36: Transformation of the IO matches into dependencies among IDs and ODS.

List_{selectedBooks} respectively, denoted by 1, 2, and 3 in Figure 3.38). It is interesting to note that the CheckTotal* in Figure 3.38 is obtained by:

1. Absorbing its $OD$ as it has no output links,

2. Resetting its AND-split to an EMPTY as it has one outgoing link only, and finally by

3. Absorbing its $OC$ as it has an EMPTY-split while its $OC$ has a XOR one.

The process of buying a list of books with this aggregated service follows the previous two scenarios. However, a particularity of this aggregated workflow is that the GetBookPrices*, Add2ShoppingCart*, and RefineBookList* tasks can be
enabled from the data-flow viewpoint by the execution of either \texttt{ChooseBooks}\(^*\), or \texttt{RefineBookList}\(^*\). Hence, a client of the aggregated service may update the list of desired books by first emptying the shopping cart, followed by the refinement of the book list, and finally by adding them to the shopping cart. Furthermore, we recall that the execution of the \texttt{GetBookPrices}\(^*\) multiple-instance task leads to executing one of its instances for each book in the list. Moreover, \texttt{GetBookPrices}\(^*\) terminates (and hence it outputs tokens) only when all its instances have finished their execution.

### 3.5.4 Fourth Example

For our last example, we add a cancellation set to the \textit{Client2} workflow, which is in charge of cancelling the purchase of a list of books at a certain timeout. The workflows to be aggregated are given in Figure 3.39.

The \textit{Client} workflow, now called \textit{Client3}, starts with the execution of the \texttt{ChooseBooks} task, as in the previous example. However, after executing \texttt{ChooseBooks}, the workflow executes concurrently the \texttt{CheckTotal} and the \texttt{Wait} tasks. Basically, the execution of the \texttt{Wait} task resumes to waiting for a certain amount of time \(t\), which
Figure 3.38: The final workflow of the service obtained by aggregating the three services in Figure 3.32.

is given as input. (Please note that we have not represented the input of Wait, as well as we shall not go into any details about the YAWL TimeService implementing the Wait task [87] as they are not crucial for the presentation of our methodology.) When the amount of time $t$ has elapsed (viz., the Wait task has finished its execution), the YAWL engine removes all tokens from the cancellation set of Wait. Hence, the Wait task is in charge of cancelling the purchase of a list of books given a time period has elapsed. In this scenario, the execution of the workflow finishes as Wait outputs a token for the End task.

The second cancellation set associated to the GetPaymentDetails task serves to cancel the Wait timer. The execution of the GetPaymentDetails task invalidates the execution of the Wait task in order to prevent the cancellation of the purchase when the Client3 workflow has outputted the credit card details and the delivery address.

Task Expansion

The particularity of this example is the usage of cancellation sets. As described in Subsection 3.4.4, if a task $X$ belongs to a cancellation set, then the Task Expansion step basically includes in the respective cancellation set all expansion dummies of $X$. 
Figure 3.39: Final example for illustrating how the aggregation copes with cancellation sets.

For example, Figure 3.40 illustrates the expansion of the four tasks of the Client3 workflow belonging to the two cancellation sets. On the one hand, the cancellation set of $Wait^*$ includes the IC/IDs and OC/ODs of the three other tasks, while the cancellation set of $GetPaymentDetails^*$ includes only the $Wait^*$ task. Furthermore, the condition in the cancellation set of $Wait$ is included into the cancellation set of $Wait^*$ as well.

Control-Flow Analysis, Data-Flow Analysis, and Contract Optimisation

The Control-Flow Analysis step does not change when dealing with cancellation sets. Consequently, the rough aggregate for this example is quite similar to the one obtained for the previous example (see Figure 3.37). The main add-on of this rough aggregated workflow consists of the two cancellation sets, as described in the Task Expansion step (see Figure 3.40). This is mainly due to the fact that the only new task of this example is $Wait$, which adds nothing to the previously obtained IO matches table\(^9\) (see Figure 3.35).

\(^9\)Please see the discussion at the beginning of this example, in which we motivate why we do not represent the input of the $Wait$ task.
As explained in the previous example, the Data-Flow Analysis adds three dummies for dealing with multiple output matches for the bookList inputs of GetBookPrices and Add2ShoppingCart, and for the selectedBooks input of RefineBookList. While the former two dummies do not lead to any changes in the aggregated workflow, it is important to note that the RefineBookList selectedBooks dummy has to be added to the cancellation set of Wait*.

Also with respect to cancellation sets, the Contract Optimisation step acts by removing dummies from cancellation sets when they are absorbed into other tasks. For example, this is the case of the ID and OD dummies of the CheckTotal* task (see Figure 3.41).

The final aggregate workflow of this example is depicted in Figure 3.42. Note that after removing all redundant dummies, the cancellation set of Wait still includes the RefineBookList selectedBooks input dummy of RedefineBookList* (denoted by 3 in Figure 3.42).

As one may have noted, there are two possible execution scenarios for this aggregated workflow. On the one hand, if the purchase of the books ends before the Wait* timer elapses, the execution behaviour of the aggregate is quite similar to the previous example. The main difference is that here the execution of the GetPaymentDetails* task leads to the removal of all the tokens in its cancellation set, and consequently to the cancellation of the timer. On the other hand, if Wait* terminates (viz., the timer elapses) before GetPaymentDetails* does, then the entire aggregated workflow locks as, for example, the ConfirmOrder* task blocks waiting for the payment details. It is important to note that the lock is due to a behaviour mismatch between the participant workflows, and not due to a flaw in the aggregation process. In Section 3.4 we showed how a reachability analysis of YAWL workflows can be employed to verify e.g., lock-freedom, while in Chapter 4 we present a YAWL-based adaptation approach for tackling behavioural mismatches.

Figure 3.40: Expanding tasks included in, or associated to cancellation sets.
3.6 Implementation

In this Section we discuss the main implementation aspects (e.g., choice of data structures, marshalling and unmarshalling of YAWL workflows, etc.) of Sator, our Java proof-of-concept prototype implementation of the core aggregation process described in Subsection 3.4.4. Furthermore, we include some words on the Java packages implementing the aggregation as well as a URL for downloading the source code of Sator.

3.6.1 Implementation Choices.

The main implementation choices were conditioned by the following aspects:

- Selection of the programming language for the implementation,

- Transposition of YAWL workflows from a XML representation into data structures, on which the aggregation can be applied,
3.6. IMPLEMENTATION

**Figure 3.42:** The final workflow of the service obtained by aggregating the three services in Figure 3.39.

- Format and acquisition of the data-flow mapping (i.e., a set of dependencies among the IOs of tasks belonging to different workflows), and

- Deployment of the data structures produced by the aggregation process into a XML file representing the aggregated workflow.

In order to ensure portability, we chose Java for the implementation of Sator. Java allowed us to import the YAWL engine code library, therefore avoiding re-implementing the “unmarshalling” (viz., transposition of XML files into data structures) and “marshalling” (viz., deployment of the data structures into XML files) phases. Furthermore, this choice has delivered two distinctive advantages:

- **Code modularity:** it has not been necessary to implement already existing solutions, thus limiting the coding work only to the aggregator, and

- **Forward compatibility** (with respect to the YAWL engine and editor): the YAWL deployment files are tied up through a XML Schema and, whenever new versions of the YAWL tools are released, this schema can be updated.
With respect to the data structures, we preferred to adopt those defined in the YAWL code library, as they are both the result of the unmarshalling process and the needed starting point for the marshalling phase. However, as future work, we plan to introduce an intermediate step to convert YAWL data structures into a set of data structures specifically optimised for the aggregation algorithm, thus making the implementation more efficient in aggregating large sets of services.

As for the format of the data-flow mapping, we chose a simple XML format, for homogeneity reasons with the rest of the input files. In Section 3.4 we argued that ontology-based matching can be applied to automatically derive the data-flow dependencies linking workflow tasks from the semantic descriptions of the services to be aggregated.

### 3.6.2 Main Implementation Solutions.

The main implementation solutions can be synthesised as follows:

- **Low-level representation of EMPTY-join/-split constructs.** The YAWL libraries represent (at low-level) EMPTY-join and -split constructs as XOR-joins and AND-splits, respectively. For a correct application of the aggregation algorithm, in order to verify at deployment time whether a join/split was initially an EMPTY one, some controls have been set up to check the number of incoming/outgoing task links. Namely, for every XOR-join/AND-split found, we mark it as EMPTY-join/-split if there exists only one incoming/outgoing task link.

- **Cancellation sets.** Cancellation sets are an important feature of YAWL. Therefore, they have been taken into account in the implementation, making them consistent in the aggregated workflow. Due to the fact that the aggregation process introduces dummy tasks in the aggregated workflow, one may not simply recreate the cancellation sets as they were defined in original workflows to be aggregated. Instead, cancellation sets are first saved without explicit re-association with a task, and then, after optimisation, once the absorption of every redundant dummy has been completed, reassigned to the corresponding task. In this process, care is taken to extend them to include dummies, if any, relative to tasks in the original cancellation set. During this operation, we take into account the new (unique) identifications, assigned both to the task associated with the cancellation set and to the tasks and conditions in the set.

- **Input/Output parameters and global variables.** A substantial difference between the high-level and low-level views of a YAWL workflow is that, at the high-level, the mapping that binds I/O parameters and net variables is not represented. These associations are defined in the YAWL deployment files representing workflows, via the `startingMapping` (relative to input parameters) and the `completedMapping` (relative to output parameters) attributes,
and consequently, they must be correctly adjusted in the aggregated workflow by taking into account the new variable identifications, as well as the new net they belong to. Moreover, in order to respect the data-flow mapping, every output parameter of a task has been mapped onto several global variables (associated to input parameters of other processes), whose identifications are given by the relative dependencies in the data-flow mapping. If an output has no dependencies in the data-flow mapping, then we map it on the new identification of the net variable originally associated with it. Net variables that are no more taken as input by any task after the re-association process are discarded.

- **OR and XOR predicates.** The XPath predicates associated with the control-flow links outgoing from tasks with OR- or XOR-splits, used to control conditional execution, are logical expressions (typically) built upon net variables. In order to deal with the new variable identifications, as well as the fact that the string used to resolve variable names also contains the parent net, the implementation includes a method to parse and “dissect” the original predicates and then to rebuild them, coherently with the new (aggregated) parent net and with the new variable identifications. Then, the predicates are associated with the respective outgoing links, following the original evaluation order and default flow.

- **Implicit conditions introduction and treatment.** Given the use of the YAWL engine code library, we had to take into account the implicit conditions, which YAWL considers at a low-level, between each two tasks linked by a control-flow link. Therefore, implicit conditions have been created during the phases of Task Expansion, Control-Flow Analysis, as well as Data-Flow Analysis. Due to the partial immutability of YAWL data structures, following to the optimisation phase we had to normalise the aggregated workflow with respect to implicit conditions, in order to first remove possible series of implicit conditions and multiple links outgoing from a single implicit condition, and second to remove implicit conditions leading to “blind alleys”, which result from the elimination of OD dummies of tasks that do not have outputs used in the aggregated workflow.

### 3.6.3 Code Structure and Code Quality Evaluation.

The implementation consists of three packages: `wsa.aggregation`, `wsa.support`, and `wsa.user_interface`. The first one contains the `AggregatedYSpecification` class, which holds the aggregated workflow and the methods relative to the aggregation phases. The `wsa.support` package contains some record classes used to pass complex data during the aggregation, and some support methods used to work out some low-level problems such as transposition of mappings between global vari-
ables and process parameters of the starting services, production of unique identifications for global variables in the aggregated workflow, and so on. Finally, the *wsa.user_interface* includes the classes concerning the GUI of the aggregator.

The source code of *Sator* is freely usable, modifiable and re-distributable under GPL license. The interested reader can download it from: http://www.di.unipi.it/~popescu/Sator_SourceCode.zip.

### 3.7 Complexity Analysis

In the following we shall informally discuss the complexity of our approach by briefly analysing the various phases involved in the aggregation process.

- **Reachability Analysis and Trace Tables.** As described in Subsections 3.4.1 and 3.4.2, the successful traces of a service are determined first by building the MRT/RG of its workflow, and then by synthesising the corresponding TT. While the algorithm for generating MRTs [96] has the same order of complexity of the algorithm for generating FRTs, unfortunately the reachability problem (also called coverability problem) for Petri nets is known to be EXPSPACE-hard [34].

  As described in Subsection 3.4.2, a TT is built by synthesising all MRT paths leading from the initial to the final marking, by considering at most once each loop in the graph. As a consequence, also the complexity of generating TTs is EXPTIME. It is however worth noting that the generation of both the MRTs/RGs and the TTs of the services to be aggregated is performed off-line, that is, it does not affect the efficiency of the overall aggregation process at query-time.

- **Service Matching.** As described in Subsection 3.4.3, given a registry containing $N$ services, this phase first looks for a set of candidate services that satisfy all the $c$ traces of the client. If no such set exists, a set satisfying $c - 1$ client traces is looked for and so on, until considering a single client trace. To satisfy a set of $x$ client traces, the construction of the MG starts with the initial node that contains the inputs needed and the outputs generated by the $x$ client traces. Further nodes are added for each service trace generating an input needed by some (not yet visited) node.

  If we assume that the total number of traces of all services is $O(N)$, the MG will contain at most $O(2^N)$ nodes, and hence the overall construction of the MG (if we consider all possible combinations of client traces) will require $O(\sum_{x=0}^{c-1}(C(c, x) \times 2^N))$, that is, $O(2^{c+N})$, or $O(2^N)$ steps in the worst case. Note however that the implementation of the service matching phase outputs one candidate set at a time (to the following aggregation phase), and hence
after the first generate&test succeeds the client does not need to wait for the generation of all other candidate sets.

- **Core Aggregation and Contract Generation.** As described in Subsection 3.4.4, this phase is performed on each set of candidate services generated by the previous Service Matching phase. Let $T$ be the number of tasks contained in the workflow representing the $S$ candidate services to be analysed. The Task Expansion step generates for each task (at most) four dummies, hence requiring $O(T)$ time, while the Control-flow Analysis connects (at most) $T^2$ tasks, hence requiring $O(T^2)$ time. The Data-flow Analysis will connect each other at most $S \times T$ tasks, hence taking $O((S \times T)^2)$ in the worst case. Finally, the Contract Optimisation step removes the redundant dummies introduced during the previous steps. As there are at most four dummies for each task, this step will take $O(T)$ time. Hence, overall the complexity of the Core Aggregation and Contract Generation is $O((S \times T)^2)$.

- **Contract Validation.** We already discussed the cost of generating the MRT/RG and the TT of a service. Although this phase currently generates the MRT/RG and the TT of the aggregated contract at query time, we reckon that the complexity of this construction can be sensibly reduced by deriving the MRT/RG and the TT of the aggregated contract directly from the MRTs/RGs and TTs, respectively, of the involved services.

### 3.8 Middleware Aspects

We recall that our methodology aims at offering the basis for the development of a platform for the inter/intra enterprise application integration that is able to aggregate services written using different service description languages while overcoming interaction mismatches. Although our focus is on describing the methodology, we shall summarise hereafter the main middleware aspects of our aggregation approach.

The high-level view of the architecture we propose for the aggregation process can be seen in Figure 3.43. We reckon that each each phase of the aggregation can be deployed as a Web service, as well as the entire aggregation process as a BPEL process orchestrating the participant (sub)services.

Some of the aggregation phases and tools can be implemented in Java and then deployed as Web services (e.g., **Core Aggregation and Contract Generation (CACG)**, **MRT/RG & TT Generator**, and so on). For example, the tests carried out with **Sator**, our Java proof-of-concept prototype implementation of the CACG (e.g., the aggregation of the services in [22], or of the services introduced in Section 3.5) show that this phase of the aggregation process can indeed be automated. Another example is the program of Wong and Zhou [98] for the automated generation of MRTs.
CHAPTER 3. WEB SERVICE AGGREGATION

Figure 3.43: Deploying the aggregation technique as a BPEL process.
Similarly to IDL interfaces for components, each service will specify a WSDL interface with the operations it provides. For example, CACG has a WSDL CACG Interface, which offers a CoreAggContractGen operation requesting one set of contracts as input and generating their aggregated contract as output. Another example is the MRT/RG & TT Generator tool implemented in Java and deployed as a Web service. Note that its WSDL interface is used both by the Service Translation and by the Contract Validation BPEL services implementing the corresponding aggregation phases.

The rest of the aggregation phases can be implemented as BPEL processes. For example, the entire aggregation process can be implemented as a BPEL process (call it CoSA) orchestrating the services of the various aggregation phases. Clients wishing to aggregate services that satisfy a certain service $C$ simply have to invoke the Aggregation operation of the CoSA WSDL interface. Since a BPEL process is exposed to its invokers through a WSDL interface, a client of the CoSA service can be another BPEL process, a Java-based application, or even a user manually invoking it (e.g., using the SOAP Client service of the ActiveBPEL suite\textsuperscript{10}). Figure 3.43 depicts a synchronous invocation of the BPEL process. (Another possibility would be to invoke it asynchronously, yet in this case the client should provide a WSDL call-back interface to the BPEL process to where the latter can send the results of the aggregation.)

The behaviour of the CoSA process follows the aggregation phases described in this Chapter. Note that each phase is executed by a synchronous invocation to the corresponding Web service. The modularity of this approach further provides us with the following advantages:

1. Each aggregation phase can be deployed as a Web service featuring several subservices. For example, the Service Translation phase can be implemented as a BPEL process orchestrating several subservices, such as a BPEL2YAWL and an OWLS2YAWL subservice, each providing a WSDL interface to the Service Translation composite service. Dually, the Contract Deployment phase can be implemented as Web service composing several subservices, such as YAWL2BPEL and YAWL2OWLS. In Chapter 5 we describe the specification of a translator of BPEL processes into YAWL workflows, and give some insights on its Java proof-of-concept prototype implementation. A distinguishing feature of the BPEL2YAWL translator is that it handles all types of BPEL activities, as well as synchronisation links, events, faults and (explicit) compensation. Furthermore, the pattern-based compositional nature of the BPEL2YAWL translator sets the basis for the development of an inverse YAWL2BPEL translator.

2. Each service implementation can be updated independently of the rest.

\textsuperscript{10}http://www.activebpel.com
3. Clients may wish to just aggregate service contracts, or they may simply wish to convert Web services from one service description language into another. The former can be achieved by invoking the CACG service, while the latter can be done in two steps, first by calling Service Translation (e.g., OWLS2YAWL), and then by calling Contract Deployment (e.g., YAWL2BPEL).

4. Finally, the implementation of the aggregation phases as well as of the entire aggregation process as Web services gives us the possibility to virtually deploy them anywhere on the Web. One possibility would be to deploy all the participant Web services on the registry-side so as to maximise efficiency. For example, Service Matching should preferably be collocated with an ontology-enriched UDDI registry (e.g., [45]) so that the service selection phase does not have to download the descriptions of the advertised services. Furthermore, the core aggregation and contract validation phases can be done on the registry-side as well, so as to minimise network traffic. The contracts and traces of the advertised services can be generated off-line and stored into “contract registries”, which can be updated either manually, or automatically at certain time intervals. A further possibility is to employ spiders to periodically download and update service advertisements residing in multiple (remote) UDDI registries. In this perspective, the aggregation service could be in principle deployed to any arbitrary Web site, that would directly access a local registry.

With respect to the run-time support for the deployment of composite BPEL processes note that we aim at generating the abstract part of the BPEL process (not to be confused with “abstract BPEL processes”), which does not specify deployment details of the BPEL process (such as the Process Deployment Descriptor information in ActiveBPEL). However, the client can either manually deploy the composite BPEL process (e.g., in the ActiveBPEL engine), or use a semi-automated business process generation tool such as the Oracle BPEL Process Manager\textsuperscript{11}.

3.9 Related Work

We start this Section with a brief introduction to Web service orchestration and choreography. Then we discuss other manual, semi-automated, and automated approaches to Web service aggregation. At the end of the discussion we try to synthesise the (comparative) advantages of our approach.

Currently, there are two major views that describe compositions of Web services: orchestration and choreography. Informally, “orchestration is akin to traffic lights where events are controlled centrally, whereas choreography is more like a roundabout, where each participant is following a prearranged set of rules.”\textsuperscript{12}.

\textsuperscript{11}http://www.oracle.com/technology/bpel/
\textsuperscript{12}http://blog.whatfettle.com/2005/02/16/choreography-vs-orchestration
An orchestration describes the (complete) interaction behaviour of a party, that is, its interaction with respect to all its business partners, and it is usually expressed as an executable process. For example, BPML [6] and BPEL [13] provide support for the definition of service orchestrations in the form of executable processes. A choreography describes the message exchanges among all the partners involved in a composition; it cannot be (completely) described from the viewpoint of one participant only. In a choreography, all services share the overall coordination of the business process. A choreography can only define how one party sees the entire collaboration. WSCI [92] and WS-CDL [99] are two languages for expressing service choreographies. Furthermore, abstract BPEL [13] processes can be thought of as choreographies since they can describe the entire business process from the viewpoint of a participant [75, 89]. Although the Web service community is currently divided trying to decide which is the best approach, we argue that they can be considered as complementary tactics, rather than rivals.

Note that our core aggregation methodology generates a specification of the composite service, which can be used to deploy a service composer that orchestrates the client and the matched services into realising the composite service. More details on this topic are given in Section 3.10.

In manual Web service composition, the requester has to browse the registry, find the desired service operations, and model their interactions into a flow structure. Most manual approaches rely on the Business Process Execution Language for Web Services (BPEL for short) [13]. BPEL is a hybrid language in the sense that it combines features from both the block-structured language XLANG and the graph-based language WSFL. BPEL enables the specification of control and data logic around a set of Web service interactions. The resulting process is exposed as a Web service using WSDL.

Papazoglou et al. [107] define the Service Scheduling Language and the Service Composition Execution language, and manually produce sequential or concurrent service compositions from simple or complex Web services wrapped as components.

Aalst and Weske [89] employ WF-nets (a form of Petri nets) for the construction of private workflows from a given public workflow (described as a choreography). The process first partitions the public workflow with respect to the organisations involved, and then it constructs the private organisational workflows as projections (viz., subclasses) of the respective public views.

Dijkman and Dumas [33] present a Petri net-based formalisation of services and a tool for the static verification of service compositions. They identify four service views and the relations among them: (1) a choreography describes the communication between the involved parties, (2) an interface behaviour describes the protocol of a service with respect to a single business partner, (3) a provider behaviour presents the protocol of a service with respect to all its business partners, and (4) an orchestration exposes the complete protocol of a service provider as an executable process.
Kuo et al. [52] argue for the use of service contracts to express the “functionality a service exposes to other services”. They compare two proposals for expressing the messaging behaviour of services: message exchange patterns and process models. On the one hand, message exchange patterns only allows for the definition of request-response or solicit-response messages (e.g., as for WSDL interfaces) and hence it cannot be used to determine whether a composite works. On the other hand, process models descriptions can be very verbose and difficult to understand (e.g., complex asynchronous BPEL processes). Consequently, they argue for the use of conditional message flows, which are basically sequences of messages guarded by boolean conditions. (This is similar to our service execution traces guarded by conditions in the preconditions set.)

Pokraev et al. [76] employ OWL [62] to verify whether a composite service satisfies a given goal. Service interoperability is checked at three levels: syntactic (viz., verify whether sent/expected data formats are compatible), semantic (viz., verifying whether the exchanged data has the same meaning for both sender and receiver), and pragmatic (viz., verifying whether the exchanged data has the same effect for both sender and receiver).

Guidi et al. [38] propose a set of process calculi for the formal definition of service communication and compositions. The service behaviour calculus composes service operations (e.g., WSDL-like one-way or request-response) through usual process calculi operators (e.g., sequential, parallel, choice). The service engine calculus defines the execution environment for services; it makes services state-full by considering BPEL-like correlation sets. Lust but not least, the service system calculus serves for defining parallel compositions of service engines.

Pankratius and Stucky [72] describe a Petri net-based formalisation of workflow compositions. Based on relational algebra operators, the authors have defined operators (viz., selection, projection, join, union, and difference) for the construction of new Petri nets from existing ones. However, the proposed methodology employs token-less Petri nets and hence it can only be employed for the structural analysis of the modelled system.

Kazhamiakin and Pistore [47] provide a data-aware approach for the verification of Web service compositions. The proposed methodology models BPEL processes as state transition systems, in which every BPEL activity leads to a transition in the system. However, the authors do not describe how to model complex process behaviour (e.g., involving dead-path-elimination and fault handling). Briefly, the approach inputs a specification of the composition and it verifies existential and universal properties over data.

Hamadi and Benatallah [39] propose a Petri net-based algebra for the definition of composite Web services. The algebra operators allow the composition of Petri nets into e.g., sequences, choices, arbitrary sequences, or iterations. The approach can be employed for the formal verification of Web service properties. However, the proposed algebra does not take into account the data-flow of Web services. Furthermore, the authors do not describe the process of translating complex Web
3.9. RELATED WORK

Semi-automated composition of services usually involves a service composition system that interacts with the requester in an iterative manner in order to obtain information about the requested service, and to construct aggregate service(s) out of the registered ones. An example of such approach is the intelligent registry with constraint matching capabilities proposed by Liang et al. [53]. The authors define a service dependency graph, where constraints may specify data dependencies as well as extra-functional properties of services. However, the accuracy of the discovery is limited by the absence of semantic information.

Bouguettaya et al. [64] model the control-flow of the desired composed service while service advertisements are described through their IOs only. The composition is done by matching requested operations with the advertised ones based on IOs and non-functional properties.

Foster et al. [36] model composite service protocols as finite state processes (FSPs) obtained from given message sequence charts (MSCs) that describe the interaction of each participant service. The validation of the composite service employs a trace-based analysis to establish whether the implemented (BPEL) process satisfies the design scenarios and whether it is lock free. The proposed methodology has been implemented as a tool for the analysis of MSCs and BPEL processes.

Charif and Sabouret [28] provide an overview of approaches for the semantic composition of Web services and propose an ontology-aware AI-based technique for the generation of service compositions. The proposed approach deals with behaviour-less services modelled through the view design language, which is an AI-based programming language. A mediator service inputs user requirements (viz., sets of service outputs) and it looks for services that are able to provide the needed requirements.

The automated composition of services has gained advance in the last years. It assumes the existence of a discovery agent that receives a service request and then it generates a structure of services/operations of some registered services based on the information provided in the request. Thakkar et al. [83] model Web services as Datalog rules. A service request is represented by domain predicates that are further unionised with the inverted service rules in order to produce a Datalog program. Then, by processing the respective program one obtains the result for the request.

Ponnekanti et al. propose SWORD [77] that also represents services as rules (i.e., LHS specifies the inputs while RHS the outputs). Such rules are processed by a rule-based system in order to derive new services.

Many approaches based on artificial intelligence techniques model the service composition problem as a planning one. Given services modelled as atomic actions and a client goal, the answer comes in the form of a plan which transforms the initial state into the requested one. For example, McIlraith et al. [63] employ an adaptation of Golog (a high-level logic programming language based on situation calculus) for the composition of Semantic Web services. The DAML-S [31] service
descriptions are translated into Prolog facts. Based on the Prolog facts and the
goal description of the user, Golog can instantiate predefined plan templates for the
composite service.

Wu describes in [104] SHOP2 – a hierarchical task network (HTN) planning
system that automatically discovers composite Web services (i.e., tasks) from a
DAML-S service registry. It does so by decomposing a task into sub-tasks until all
sub-tasks can be performed directly.

Traverso et al. [84] use non-deterministic transition systems to model both ser-

vices and client. Given a set of advertisements and a global goal, their algorithm
outputs a plan which coordinates services so as to satisfy the goal.

Berardi et al. [11, 12] model service and client behaviour as finite state transition
systems in which a transition abstracts the IO messages and operations. Roughly,
the composition synthesis looks for service transition systems that can be composed
to construct the given client request. Consequently, the output is automatically
generated by delegating the requested actions to ones of the advertised services.

Yan et al. [106] model Web service compositions as AI planning problems and

employ genetic algorithms techniques to optimise the planning results. A downside
of their approach is that they do not describe how (complex) business processes (e.g.,
BPEL services) are to be modelled as planning problems. However, a downside of
planning (besides computational cost) is that representing the goal is difficult and
error-prone.

In the following we describe several other approaches for the formalisation of
Web service properties (e.g., lock freedom), discovery and interaction.

The approach of Massuthe and Wolf [61] employs automata for matching services
and represents the communication behaviour as labels to transitions of the service
automaton. Composite transition systems represent service interactions. The inter-
action is restricted to two parties (one service provider and one service requester).
The methodology formalises deadlocks and weak termination, as well as compliant
sub-automata. The operating guideline of a service is defined as the characterisation
of all properly interacting partners of the service. The authors argue that brokers
can use the operating guideline of a provided service so as to match a service request.

Lohmann et al. [55] provide a formal framework for the analysis of BPEL pro-
cesses. BPEL processes are translated into oWFNs (open workflow nets), and service
interfaces are expressed as sets of IO places. oWFNs further have one initial place
and several final places. The authors employ operating guidelines to check whether
two oWFNs interact successfully.

In [60] Massuthe et al. argue that providers should publish the operating guide-
line of a service modelled by an oWFNs. Then, the discovery process reduces to
comparing the operating guideline of the provider service with the behaviour of the
client oWFN (viz., whether the latter is a subtree of the former). The behaviour of
a service is obtained through a reachability analysis of the service’s oWFN.

In [56] Lohmann et al. extend [60] and employ the operating guidelines for the
3.9. RELATED WORK

formalisation of deadlock-free interactions between finite-state services.

In [58] Martens defines the notion of usability of business processes and then he derives notions of compatibility, simulation, and equivalence between business processes. The approach is based on a Petri net formalisation of BPEL processes. From the Petri net representation of the business processes Martens defines the communication graph (c-graph) of a BPEL-process, that is, its externally visible behaviour. Next, usability graphs (u-graphs) are defined as usable behaviour of business processes. Roughly, u-graphs are non-empty subgraphs of c-graphs obtained by removing edges that start at visible nodes and their successors, where visible nodes represent states in which the process is either waiting for an input, or it has terminated.

Furthermore, in [59] Martens employs c-graphs for matching business processes based on their behaviour. Basically, the core of the discovery process checks whether the behaviour (viz., c-graph) of the provider service simulates [58] the behaviour of the requested one.

A common downside of these approaches is that they tackle only acyclic service automata [55, 61], oWFNs [55, 60], or finite-state c-graphs [58, 59]. Furthermore, [55, 56, 58, 59, 60, 61] do not cope with data-flow aspects and do not employ ontologies. Finally, [59, 60, 61] match single services; they do not tackle the generation of aggregates that can match client requests.

Roughly, the above mentioned approaches present somewhat similar approaches that can be exploited for the discovery and composition of Web services. However, we argue that our approach copes with the data-flow aspects of services, and employ ontologies for service matching. Furthermore, our reachability analysis employs MRTs [96] provided reachability graphs are infinite due to cycles in the service workflows.

Several reviews accurately describe current trends in Web services composition. In [49], Srivastava notes the two main trends in Web service composition: “Web Services in the Semantic Web: RDF/DAML-S + Golog/Planning” (i.e., the Semantic Web approach) vs. “Web Services in Industry: WSDL + BPEL” (i.e., the industrial approach). In [2], Aalst et al. present a comparison of BPEL, XLANG [80], WSFL [101], BPML [6], and WSCI [92]. They show the trade-off between block-structured languages (e.g., XLANG, BPML, and WSCI) and graph-based languages (e.g., WSFL). An interesting comparison between BPEL and DAML-S is provided by [54], while another one between BPEL and WSCI is given in [108]. An analysis of Web service composition languages providing another comparison of BPEL, XLANG, WSFL, BPML and WSCI (with an accent on analysing BPEL) can be found in [97]. Yushi et al. [26] overview and compare BPEL [13], BPML [6], and WS-CDL [99] against several generic requirements that composition languages should possess (e.g., exception handling, event handling, transactions and compensations).

It is worth observing that our approach is the first — at the best of our knowledge — to provide the following features in a single framework: (a) it is a fully
automated approach capable of generating service aggregations that fully/partially satisfy behavioural queries, (b) it supports both service selection and aggregation at the level of traces (and not at the entire service level), (c) it relies on service contracts and traces that can be computed off-line, (d) it can be exploited to discover and aggregate services written in different languages, and to generate multiple deployments of the aggregated contract given that it relies on intermediate YAWL descriptions of the behaviour of services, as well as (e) it can be used to locate, aggregate and adapt BPEL processes, as it straightforwardly integrates with the adaptation process described in Chapter 4, and with the BPEL2YAWL translator presented in Chapter 5.

Finally, it may be worth mentioning the relation between our methodology (to prove properties) and model checking. Model checking is a method to algorithmically verify whether the model of a formal system satisfies a formal specification. The model is usually expressed as a transition system in which atomic propositions are associated to each node, and the specification is often written as temporal logic formulas. In our setting, the model is represented by the MRT/RG, where each node represents a state of the system and has an associated condition, and properties are verified by checking conditions over the MRT/RG. The verification of lock-freedom, for instance, reduces to checking that the MRT/RG of the analysed service does not include deadlock markings (viz., non-final nodes without outgoing links). From the abstract complexity viewpoint, our approach inherits the EXPSPACE complexity of traditional model-checking techniques. It is however worth noting (as already mentioned in Subsection 3.4.4) that the generation of both the MRTs/RGs and the TTs of the services to be aggregated is performed off-line, and that the generate & test coordination of the service matching and aggregation phases sensibly lowers the concrete complexity of the approach.

3.10 Discussion

In this Chapter we described the process of aggregating Web services with the goal of satisfying behavioural client requests, which is part of our aggregation and adaptation methodology that aims at integrating business processes written with different service description languages (e.g., BPEL [13] or OWL-S [71]) by suitably overcoming semantic and behaviour mismatches.

A key ingredient of our methodology is the notion of service contract consisting of a signature, an ontology description, as well as a behaviour specification expressed through an (abstract) formal language. Contracts are the basis for linking services through data-flow dependencies, as well as for overcoming signature and behaviour mismatches. They also pave the way for composing services written in different languages, and for multiple deployments of the composite service. A good candidate for a language to describe the ontology information is OWL [62], and parameter matching algorithms such as [73] can be employed to match service traces, as well
as to derive the data-flow mapping among the services to be aggregated. Furthermore, the client can provide sets of equivalent parameter types belonging to different parameter ontologies. We chose YAWL \cite{87} for expressing the behaviour of a service contract mainly due to the fact that is a formal language defining twenty of the most common workflow patterns. Furthermore, YAWL gives the possibility of expressing complex control- and data-flow.

We argue that each service should advertise its service contract. It is important to note that their generation can be done off-line and hence it is not a burden for the aggregation process. The MRT \cite{96} is a very useful tool that can be successfully employed for analysing service properties such as reachability or lock-freedom. Due to the fact that a MRT can be equivalently represented as a RG for bounded workflows, as well as for the simpler and more compact notation of the latter, we chose to present here the application of our methodology using the RG. However, the usage of the MRT is slightly more complex due to the usage of the $\omega$-numbers to cope with workflow unboundness. From the MRT/RG we extract the successful execution traces of a service, which are summarised in entries of the TT. By inspecting such entries we can easily determine which tasks are to be executed, which inputs are needed, as well as which outputs are generated for an execution trace.

The aggregation algorithm firstly generates candidate sets of services by matching successful traces of the advertised services with successful traces of the client service. A candidate set together with the matching traces of the client corresponds to a closed workflow from the data-flow point of view. For each candidate set we generate the contract of the aggregated service by suitably constructing its control- and data-flow. Basically, the former is achieved by invoking all component services in parallel, while the latter is achieved by translating the data-flow mapping obtained by matching task parameters into dependencies among workflow tasks. In order to verify whether the aggregated service satisfies a successful client trace, we generate the TT of the aggregate. The goal resumes to checking whether the tasks executed by the respective client trace are executed by at least one of the successful execution traces of the aggregate.

For each satisfied client trace our algorithm gives a \textsc{YES} or \textsc{MAYBE} answer. While for the former the client is always satisfied, for the latter the fulfilment of the client trace is subject to conditions used for managing the control-flow of the composed services. We say that the aggregation is \textit{successful} (or that the client is \textit{fully satisfied}) if all client traces are satisfied, \textit{partially successful} (or that the client is \textit{partially satisfied}) if some, yet not all client traces are satisfied. If no client traces are satisfied then the respective candidate set cannot be used to fulfil the request and hence we have a \textit{failure}. The output of our aggregation algorithm is a list of successful service compositions ordered by the number of unconstrained satisfied client traces (i.e., \textsc{YES} answers).

One main line for future work consists in deploying the successfully aggregated contracts as composite services. In short, an aggregated contract specifies how the
composition works. In this view, the composite service acts as a service composer that orchestrates the participant services. The deployment of the composer involves generating dual-views (see example below) of the participant services. Similarly to e.g., a BPEL process, the orchestrator, when invoked by its client, will suitably call the participant services, and hence it will compute the result of the composition. For example, a control-flow dependency from a task $P$ (of a contract $C$) to another task $Q$ (of another contract $S$) in the aggregated contract is reflected in the orchestrator service as:

1. The orchestrator will first wait for the output(s) of $P$, then
2. It will store the respective output(s) for later use, and
3. When $Q$ will be executable from the control-flow viewpoint, the orchestrator will provide the stored output(s) of $P$ as the needed input(s) of $Q$.

since the respective dependency states that “$P$ should be executed first, and its output(s) should be used for the execution of $Q$, that will take place at a later moment.

To be more concrete, assume that $P$ translates an asynchronous BPEL invoke activity that calls an operation $op$ and it passes it an input message $m$. Assume further that $Q$ translates a BPEL receive activity, and for simplicity, that it waits for an invocation on the same operation $op$ and with the same input message $m$. Since $C$ and $S$ do not communicate directly (e.g., assume $P$ and $Q$ address different WSDL port types), the BPEL orchestrator process should:

1. Define a receive activity matching the invoke of $P$ (the receive is the dual of the invoke, and vice-versa), then
2. Store the message $m$ through an assign activity in a variable $tmp$, and finally
3. Define an asynchronous invoke activity matching the receive of $Q$.

Furthermore, the first two activities can always be placed in a BPEL sequence activity, which is to be linked then through a synchronisation link to the third activity.\footnote{As we described in Subsection 3.4.4, the core aggregation process basically places in parallel the workflows of the services to be composed; this always leads when the aggregate is deployed e.g., as a BPEL process to a flow structured activity, hence it is always possible to model the task dependencies as synchronisation links.} In this way, the orchestrator will first execute (1.) and immediately after (2.), followed by (3.) at a later moment in time, when the execution of $Q$ will reach the receive.

For more information on generating the dual-views of a service please see Section 4.2.
Chapter 4

Web Service Adaptation

In this Chapter we present an approach that tackles the automated adaptation of services with the purpose of satisfying both, functional, and behavioural client queries.\(^1\)

Three strong motivations for adapting services are the need to develop adapters for service composition, for ensuring backwards compatibility of the new versions of services, as well as the need to develop adapters for each class of clients a service may have.

The adaptation process is part of the general methodology for aggregating and adapting services, which concerns the location of (compositions of) services that can fulfil a client request. A key ingredient of the adaptation is the use of service contracts (previously introduced in Subsection 3.3) including WSDL signatures and YAWL behaviour, where YAWL is used as intermediate (formal) language to provide a (partial) description of a service behaviour. Immediate advantages of using such an abstract language are the possibility of adapting services written in different service description languages, multiple deployment of the adapter as a real-world service, as well as developing formal analyses and transformations independently of the different languages used by providers to describe the behaviour of their services. As a result, the adaptation approach straightforwardly integrates with the contract-based aggregation of Web services presented in Chapter 3. On the one hand, service compositions could be adapted so as to (fully) satisfy client requirements, while on the other hand, services could be first adapted and then aggregated into successful service compositions.

Given a service registry \(R\) and a client query \(Q\), if a (composite) service \(S\) produced by the aggregation process (described in Chapter 3) is unable of (fully) satisfying \(Q\) due to behavioural mismatches between the two, the adaptation process plugs-in as an attempt to mediate their interaction. However, in a broader view, the adaptation takes as input a service contract \(S\) and a client query \(Q\), and it deals with adapting \(S\) so as to satisfy \(Q\). Depending on the type of the request, the

\(^1\)Previous versions of the adaptation process have appeared in [20, 21, 23].
adaptation behaves in one of the following two ways:

- **Functional adaptation.** If $Q$ is expressed as a black-box that requests services with certain IOs (viz., a service contract without behaviour information), the adaptation process generates, starting from $S$, an adapted service $S'$ whose behaviour is compliant with the request $Q$ (whenever possible). Informally, the workflow of $S'$ corresponds to a refined behaviour of the initial service $S$ that enforces the needed adaptation.

- **Behavioural adaptation.** If $Q$ specifies behaviour information as well (i.e., $Q$ is given as another service contract), the adaptation attempts to generate an adapter $A$ that can be used as a service-in-the-middle between $S$ and $Q$ as a way to overcome their interaction mismatches.

In order to better illustrate the adaptation approach, we shall describe the two forms of the adaptation process in two separate Subsections. On the one hand, Section 4.1 describes the functional adaptation that, given a (composite) service whose behaviour does not comply with a functional (viz., black-box like) client request, it generates the contract of the adapted service basically by pruning parts of the original workflow that are redundant with respect to the query, and then by suitably re-linking the remaining workflow tasks.

On the other hand, in Section 4.2 we present the applicability of the behavioural adaptation process (and of the entire methodology) by explaining how the adaptation can be employed for the automated generation of adapters capable of solving behavioural mismatches among interacting business processes. In short, the behavioural adaptation process inputs two communicating BPEL processes whose interaction may lock, and it outputs a BPEL adapter process (if any) that lets the two processes successfully interact. The adaptation first translates the BPEL descriptions of the input processes into corresponding YAWL workflows. Then, it generates the execution traces of the two workflows, and of their dual workflows. The execution traces of the adapter are then constructed from the execution traces of the dual workflows. Finally, the adapter execution traces lead to the generation of the YAWL workflow of the adapter, and then to its deployment as a BPEL process.

The features of our adaptation approach are:

1. It is a fully automated approach capable of generating service contracts tailored to functional and behavioural client requests. For example, it can be employed to automatically synthesise full/partial BPEL adapters from two input BPEL processes,

2. It generates the YAWL workflow of the adapter and/or of the adapted service, which can be used to check properties (e.g., lock-freedom, reachability, liveness) of the interaction with the client service, or of the adapted service,
3. It supports both service location and adaptation at the level of service execution traces (and not only of entire services),

4. It is amenable to efficient implementations, as it relies on the inspection of execution traces that can be generated off-line,

5. It can be exploited to locate and adapt (compositions of) services written in different languages, and to generate multiple deployments of the adapter and/or adapted contract – given that it relies on service contracts defining YAWL descriptions of the behaviour of services, and

6. It straightforwardly integrates with the service aggregation approach (described in Chapter 3) into a methodology for the aggregation, adaptation, and verification of services, since both are based on service contracts that make use of YAWL workflows to represent service behaviour.

The Chapter is organised as follows. Section 4.1 focuses on the customisation of services to functional client requests while, in Section 4.2 we describe the technique of adapting services to behavioural client queries. Section 4.3 analyses the complexity of the two adaptation techniques. Furthermore, Section 4.4 briefly discusses related work, and Section 4.5 presents some concluding remarks.

4.1 Functional Service Adaptation

The Section starts with the presentation of a simple motivating example (Subsection 4.1.1). Subsection 4.1.2 briefly recalls how the methodology matches a candidate service, and how it verifies whether the candidate fulfils the query. The following Subsection 4.1.3 presents the core of the functional adaptation process, that is, the generation of the adapted service contract. In Subsection 4.1.4 we informally prove the correctness of the functional adaptation technique.

4.1.1 Motivating Example

Consider a registry of service contracts containing the example in Figure 4.1. The Clothing Shop workflow describes a service that sells shirts and trousers. When invoked, the workflow firstly executes the Choose Item task, for which the client has to provide the item she is interested in (either “shirt” or “trousers”). The execution continues with both Choose Manufacturer and Delivery Information due to the AND-split of Choose Item. Choose Manufacturer inputs the desired designer and, based on the item to be bought, it enables only one of the following two tasks: Choose Shirt if the requested item is a shirt, or Choose Trousers otherwise. Please note its XOR-split and the predicates annotating the respective control-flow links. Both tasks input the type (e.g., “T-shirt” or “jeans”), size (e.g., “M” or “33”), and colour (e.g., “black”) of the requested item. Delivery Information is a task.
that inputs the address intended for delivery. Such information may be submitted by the client at any moment of the purchase (after choosing the item yet prior to the payment). Delivery Information outputs the shipping costs and the estimated delivery time. Last but not least, Finalise Buy employs an OR-join in order to wait for an item to be chosen, as well as for the delivery information to be available. It inputs a credit card number and it generates the purchase receipt. Please note that, in order to keep the example simple we omit scenarios in which, for example, a desired item is not available, payment issues, and so on.

Let us assume now that a client desires a service that only sells trousers. She might search one by issuing the following query:

- **inputs**: \{jeans, designer, address, size, dark-blue, cardNumber\},

- **outputs**: \{shippingCosts, receipt\}.

For simplicity we shall assume that both, query and service contracts, use the same parameter ontology and that there is an exact/plug-in/subsumes match [73]
between the following parameter pairs\(^2\): jeans and clothingItem, jeans and trouser-
sType, designer and manufacturer, address and deliveryAddress, size and trousers-
Size, dark-blue and colour, cardNumber and cardNumber, shippingCosts and ship-
pingCosts, and finally receipt and receipt.

It is easy to see that the service does not match perfectly the given query, in the
way that the service requires more inputs than those specified in the query (viz.,
shirtSize and shirtType). However, as we shall see in the following Subsection, the
service has two possible execution traces, one that sells trousers and that matches
completely the client request, and another one that sells shirts. The functional
adaptation technique describes how the Clothing Shop service can be in principle
adapted so as to inhibit its capability of selling shirts, and convert it into a “trousers
shop” service.

### 4.1.2 Matching the **Clothing Shop** Service

In this Subsection we briefly recall how the methodology matches the Clothing Shop
service from a registry of advertised service contracts, if we assume the previous
client request.

#### Deriving the Trace Table of the **Clothing Shop**

We recall that each entry of the trace table (TT) of a workflow describes a suc-
cessful execution trace, which consists of a set of triples of the form \(\langle Preconditions, Needed Inputs, Generated Outputs \rangle\), where \(\text{Preconditions}\) represents the set of data
and control constraints that must be satisfied to be able to successfully execute the
workflow, in that execution trace. \(\text{Needed Inputs}\) and \(\text{Generated Outputs}\) are the set
of inputs requested and outputs generated, respectively, by the tasks executed in
the respective trace.

In Chapter 3 (Section 3.4) we briefly described the method of generating the TT
of a workflow from a reachability graph (RG) or modified reachability tree (MRT).
The RG and the MRT are to be obtained from the YAWL workflow augmented with
explicit conditions. For example, Figure 4.2 illustrates the Clothing Shop workflow
augmented with explicit conditions. \(C_i\) and \(C_o\) denote respectively, the input and
output condition of the workflow. It is interesting to note that, due to its OR-join,
Finalise Buy is enabled if and only if \(C_7\) and either \(C_5\) or \(C_6\) contain tokens. Hence,
it cannot be enabled when only \(C_7\) contains a token and before a token arrives at
\(C_5\) or \(C_6\). Furthermore, note that tokens cannot be added to both \(C_5\) and \(C_6\) during
a workflow instance due to the XOR-split of the Choose Manufacturer task.

Figure 4.3 shows the RG corresponding to the Clothing Shop workflow in Fig-
ure 4.2. For example, the arrow from the initial marking \(C_i\) to the marking \(C_1 + C_2\) is

---

\(^2\)The first parameter in a pair belongs to the query, while the second to the **Clothing Shop**
service.
labelled as Choose Item. This is to be read as: “From the initial marking containing a token in $C_i$ only, the Choose Item task is enabled by consuming the token in $C_i$, and by firing it produces a token in $C_1$ and another in $C_2$”. The markings circled in Figure 4.3 denote markings in which Finalise Buy cannot fire as it expects one more token to be placed in one of its empty input places.

The process of generating the TT for the Clothing Shop looks in the RG/MRT of the workflow for all paths (i.e., traces) $p$ originating in the initial marking and ending in the/a final marking. The preconditions set for $p$ is given by the set of all conditions (viz., places) in the markings of $p$. The set of needed inputs is obtained by taking the inputs of all tasks labelling arcs of the path $p$. Similarly, the set of
generated outputs consists of the outputs of all tasks labelling arcs of the path $p$.

Table 4.1 illustrates the two possible execution traces of the Clothing Shop service. $T^1_{CS}$ corresponds to a successful execution trace in which the client buys a shirt, while $T^2_{CS}$ corresponds to a successful execution trace in which the client buys a pair of trousers.

| Clothing Shop | $\{T^1_{CS}, T^2_{CS}\}$: $\langle\{C_i, C_1, C_2, C_3, C_5, C_7\}, \{\text{clothingItem, } \text{manufacturer, deliveryAddress, shirtSize, shirtType, colour, cardNumber}\}, \{\text{shippingCosts, estimatedDeliveryTime, receipt}\rangle, \langle\{C_i, C_1, C_2, C_4, C_6, C_7\}, \{\text{clothingItem, } \text{manufacturer, deliveryAddress, trousersSize, trousersType, colour, cardNumber}\}, \{\text{shippingCosts, estimatedDeliveryTime, receipt}\rangle. |

Table 4.1: TT for the Clothing Shop service.

**Trace Table Compatibility Check**

In Subsection 3.4.3 we described the general process of selecting candidate sets of services that may be useful for satisfying a client request. Roughly, the matching process builds a Matchmaker Graph (MG) by matching the IOs of the query traces with the IOs of the service traces. The nodes in the graph hold sets of “needed inputs” and “generated outputs”. The initial node of the graph is given by the IOs of the query, while final nodes have the particularity that the set of needed inputs is included in the set of generated outputs. Furthermore, graph arcs are labelled by service traces. Consequently, candidate sets are obtained by considering the services labelling paths in the graph that start at the initial node and that end at a final node.

It is important to note that the output of the service matching phase is a set of services that collectively generate all the data that the query requests as input, and dually, it inputs at most all the data that is provided as input by the query. For example, given a client query specifying an input $I$ and an output $O$, the service matching phase locates (sets of) services that take $O$ as input and generate $I$ as output. In other words, the service matching finds services that could be composed with the client service, and not (composite) services that provide the same IOs as the client one. Still, as we shall see in Section 4.2, the latter behaviour of the methodology can be obtained by providing as input a dual of the client service.

The functional adaptation process described here concerns the discovery of (composite) services that can be adapted so as to provide the same functional description as the client request. For the previous example, the output of the functional adaptation process consists of (composite) services that take $I$ as input, and provide $O$ as output.

In order to match the requested query with the TT of a given service, the methodology first express the query as a simple service contract whose workflow consists of one task only, which is linked to the input and output conditions. However, in
order to locate services that offer the same functional description as the query, the inputs and the outputs of the query task correspond to the outputs and inputs, respectively, of the client request. Figure 4.4 presents the workflow, the RG and the TT for the query described in Subsection 4.1.1.

The MG one obtains for our example is given in Figure 4.5. The initial node of the MG contains (as previously mentioned) the outputs and the inputs of the client request as needed inputs and generated outputs, respectively. The node on the bottom-left side of the picture is obtained by considering the $T_1^{CS}$ trace of the Clothing Shop service, which sells shirts. The respective node is not a final node because its set of needed inputs is not contained in its set of generated outputs.

The MG node on the bottom-right side of the picture is given by taking into account the $T_2^{CS}$ trace of the Clothing Shop service, which sells trousers. Note that this node is a final node since $\{\text{shippingCosts, receipt, clothingItem, manufacturer, deliveryAddress, trousersSize, trousersType, colour, cardNumber}\} \subseteq \{\text{jeans, designer, address, size, dark-blue, cardNumber, shippingCosts, estimatedDeliveryTime, receipt}\}$. Consequently, the methodology selects the Clothing Shop service as a candidate.

---

3Recall that set operations (e.g., union and inclusion) are ontology-aware.
set for fulfilling the client request. (Note that in this case we say that the $T_{CS}^2$ trace of the Clothing Shop is compatible with the request.)

It is important to note that the preconditions set constraining $T_{CS}^2$ into satisfying the request is $\{C_1, C_1, C_2, C_4, C_6, C_7\}$. In Subsection 3.4.2 we showed how the preconditions set can be translated into a logical expression constructed from the YAWL predicates of the workflow. For instance, the above preconditions set constraining $T_{CS}^2$ into satisfying the request can be expressed as "(clothingItem = 'trousers') or (not clothingItem = 'shirt')" (due to the $C_4$ condition).

### 4.1.3 Contract Generation

Assume that the client wishes to have a deployment of a service that strictly satisfies queries of the type she has issued. In other words, she wants a service, say Trousers Boutique, that only sells dark-blue jeans made by a certain designer. This Subsection describes the core of the functional adaptation process, which achieves that by generating starting from the Clothing Shop service the contract of a service that sells only trousers.

Consider a (composite) service $S$ produced by the aggregation process described in Chapter 3. If all traces of $S$ are compatible with the request, there is no need for adapting the behaviour of $S$, which makes the output of the methodology. However, if $S$ has at least one execution trace that is compatible with the request, as well as other execution traces that do not comply with the request, the functional adaptation process employs a "pruning" technique in charge of removing the unwanted behaviour of $S$. This process is done in two steps:

1. First, for all traces $T$ of $S$ that are compatible with the query, we individuate the tasks of $T$ as workflow tasks having as input places in the preconditions set of $T$. For instance, the preconditions set $\{C_1, C_1, C_2, C_4, C_6, C_7\}$ corresponds to the set of tasks $\{\text{Choose Item}, \text{Choose Manufacturer}, \text{Delivery Information}, \text{Choose Trousers}, \text{Finalise Buy}\}$. We call redundant all other workflow tasks. This step is to ensure that we will not remove tasks (see below) needed for the successful termination of the service’s execution. For our example, (only) the Choose Shirt task is redundant as its input condition, $C_5$ is not contained in the preconditions set of the $T_{CS}^2$ trace, which is (the only trace) compatible with the query.

2. Then, we duplicate the contract of the original service $S$ into $S'$, which we modify by cancelling redundant tasks (if any) and suitably redirecting the workflow links. Workflow redirection is necessary to ensure that the workflow of the new service $S'$ is consistent with the traces satisfying the query. This is achieved by adding to the workflow of $S'$ a task, Absorb Tokens, and by directly connecting it as output of the input condition of the workflow. Furthermore, the adaptation process adds a condition as output of Absorb Tokens, which
is to be taken as input by all tasks that originally were linked as outputs of the input condition of $S$. The role of the Absorb Tokens task is to restart the workflow, by first cancelling all the tokens in the workflow, and then by producing a token for its output condition. Then, all initial control-flow links that point at a redundant task are redirected to the XOR-join of Absorb Tokens. Furthermore, all redundant tasks together with their outgoing links are removed from the workflow. Finally, service outputs that are not requested by the query are hidden (see below).

For instance, the adaptation scenario for our example yields the workflow presented in Figure 4.6.

We recall that a YAWL service is a workflow specification that consists of one or more extended workflow nets [87] – one of which is the starting net. Variables are defined at both workflow and task levels, and the data-flow is specified by binding parameters of the workflow net and of its tasks. In Figure 4.6 one may see that the Trousers Boutique service is made of one workflow net that contains the original tasks of the Clothing Shop workflow except the redundant Choose Shirt task. As previously indicated, the task Absorb Tokens together with its output condition have been added at the beginning of the workflow, and the original control-flow link leading at Choose Shirt has been redirected towards Absorb Tokens. The inner dashed portion of the workflow in Figure 4.6 delimits the cancellation region [87] associated with the Absorb Tokens task. As a consequence, whenever Absorb Tokens is executed, all tokens enabling tasks in its cancellation region are removed.
4.1. FUNCTIONAL SERVICE ADAPTATION

If a client of the *Trousers Boutique* service requests a *shirt*, then *Choose Manufacturer* will enable *Absorb Tokens* for execution and further on *Choose Item*, which will ask the client for another input. Moreover, *Absorb Tokens* will clear remaining tokens in the workflow so as to avoid (possibly) multiple executions of the other tasks in the workflow (e.g., *Delivery Information*).

Outputs that are not desired by the requester are hidden by mapping them to local net variables.\(^4\) This is the case for the *estimatedDeliveryTime* output of the *Delivery Information* task.

4.1.4 Informal Proof of Correctness

In the following we informally prove the correctness of the functional adaptation technique. The functional adaptation matches workflows \(S\) that satisfy a given (functional) query \(Q\). In other words, it checks if there exists at least one trace \(T_S\) of \(S\) that “requires less inputs than the query inputs, and that generates the query outputs (and possibly more)” . Given that, it eliminates redundant tasks, and it redirects the workflow into producing the adapted workflow \(S'\). Hereafter we show that for each trace \(T_S\) that satisfies \(Q\) there exists a trace \(T'_{S'}\) of \(S'\) that satisfies \(Q\).

**Proof.** Similarly to the aggregation process, the proof is by construction. Let \(t_1\) be the task of \(S\) that is executed first in the trace \(T_S\). Now, \(t_1\) is not a redundant task of \(S\) because \(t_1\) belongs to a trace that satisfies \(Q\). Consequently, \(t_1\) is not eliminated by the Contract Generation phase, and hence \(t_1\) is a task of the adapted workflow \(S'\). Since \(t_1\) is the first task to be executed in \(T_S\) it means that it can be executed by a token placed in the input condition of \(S\). Hence, there exists a trace \(T'_{S'}\) of \(S'\) that executes \(t_1\) (after executing the *Absorb Tokens* task – the only task added to the adapted workflow \(S'\) by the functional adaptation). This is due to the fact that \(t_1\) is an output of the *Absorb Tokens* task of \(S'\), and hence a token placed in the input condition of \(S'\) enables the *Absorb Tokens* task, that leads to the execution of \(t_1\).

Assume now \(\{t_1, \ldots, t_{k-1}\}\) tasks of \(T_S\) such that \(\{t_1, \ldots, t_{k-1}\}\) are also executed by \(T'_{S'}\). We prove that if \(t_k\) follows \(\{t_1, \ldots, t_{k-1}\}\) in the execution of \(T_S\), then \(t_k\) can also be executed in \(T'_{S'}\) after \(\{Absorb\ Tokens, t_1, \ldots, t_{k-1}\}\). Since \(t_k\) can be executed in \(T_S\) after executing \(\{t_1, \ldots, t_{k-1}\}\) we deduce that all the control-flow dependencies of \(t_k\) are met by the execution of \(\{t_1, \ldots, t_{k-1}\}\). In other words, \(\{t_1, \ldots, t_{k-1}\}\) produce all input tokens necessary for the execution of \(t_k\). Hence, we can choose \(t_k\) as the next task to be executed in \(T'_{S'}\) because \(T'_{S'}\) executed \(\{Absorb\ Tokens, t_1, \ldots, t_{k-1}\}\). Consequently, we proved by induction that all tasks \(t_k\) of \(T_S\) are also tasks of \(T'_{S'}\) (actually \(T'_{S'}\) executes *Absorb Tokens* followed by all tasks of \(T_S\)). Since *Absorb Tokens* does not require any inputs we obtain that \(T'_{S'}\) satisfies \(Q\).

\(^4\)The reason why we do not remove the undesired outputs is that, for example, the task generating them may correspond to a YAWL task that invokes a WSDL service, whose execution requires a proper mapping between its parameters and the ones of the YAWL task.
4.2 Behavioural Service Adaptation

This Section describes a technique of adapting services to behavioural client queries. As previously mentioned, we illustrate this form of adaptation by showing how to generate (BPEL) adapters that allow two interacting (BPEL) business processes presenting behavioural mismatches to interact successfully. Subsection 4.2.1 introduces a simple motivating example, while Subsection 4.2.2 describes the main phases of the adaptation, with a focus on the generation of the adapter, and validation of the adapted service, while including bird’s-eye views of the processes of service translation and adapter deployment. Subsection 4.2.3 informally proves the correctness of the behavioural adaptation technique.

4.2.1 Motivating Example

Consider the following two interacting BPEL processes: Command Centre (CC) and Mars Explorer (ME). The former provides a Web service interface for the assignment of exploration tasks. The latter is a Web service interface to the robot performing the tasks. Hereafter we present a simplification of the two BPEL processes (e.g., in order to express the message exchanges we simply use service names instead of partnerLinks and portTypes). Although fairly simple, the example illustrates various interactions among services. On the one hand, CC communicates with its client, as well as with the ME service. On the other hand, ME interacts with CC (viz., its client), as well as with the Logger and the Explorer services.

The CC service\(^5\) first receives the task information from its client. It then logs in with the ME, to which it forwards the location and the job details. It waits next either a report or an error message from the ME. In the former case, it first receives the job id from the ME, then it closes the connection with the ME, and finally, it forwards the report to the client. In the latter case, it first logs out from the ME, and then it replies to the client with the error message.

\(^5\)“Process” and “service” will be used interchangeably to denote BPEL processes.
4.2. BEHAVIOURAL SERVICE ADAPTATION

The ME service starts by waiting for the CC to log in, to which it sends immediately the job’s id. It receives next from the CC the job description and the location of the exploration site. In order to carry out the task, the ME first validates the coordinates (e.g., by checking previous exploration logs) and moves the robot to the respective location by (synchronously) invoking the Logger Service (LS), and then, it delegates the Explorer Service (ES) for the actual execution of the job (again, through a synchronous invocation). If the latter two invocations return successfully, the ME generates the final report, sends it to the CC, and waits for the CC to log out. Note that, although not represented in the example, the invocations to the LS and to the ES may return a “Task Error” fault. (This information has to be specified in the WSDL file(s) defining the respective operations). In that case, the ME service catches the fault, forwards to the CC the error, and finally, it waits for the CC to close the connection.

It is easy to see that the two services, CC and ME, cannot successfully interact because of mismatches between their behaviour. Immediately after the login information exchange, while the CC sends the location of the exploration site to the ME, the ME sends the job id to the CC. Furthermore, the CC first sends the location, and then the details of the job to the ME, which expects them in the reversed order. A further mismatch is the fact that, while the CC expects the job id only when the exploration is successful, the ME always sends it, and moreover, at a different moment.

One may think of several scenarios for selecting the two services. For example, they could have been (manually) selected by the developer from a registry of services, and then fed in as input of the core aggregation process. Another possible scenario is that the developer had submitted the (contract of the) ME service as input of the aggregation and adaptation methodology, and that the service matching phase located the (contract of the) CC service. However, for all such scenarios the aggregation of the two services fails due to the before mentioned mismatches in their communication protocols.

We recall that the adaptation process can be plugged with the aggregation technique, provided the latter process fails to produce a (composite) service that (fully)
satisfies a client request. Furthermore, the adaptation can also be seen as a stand-alone technique that can be employed for overcoming protocol mismatches among interacting Web services. In the following we describe a behavioural adaptation process that is able to generate adapters (if any) for two Web services that present interaction mismatches. In particular, based on the example introduced in this Section, we show how the adaptation can be used to automatically generate BPEL adapters that overcome behavioural mismatches between interacting BPEL processes.

4.2.2 Adaptation Phases

The behavioural adaptation process inputs two communicating BPEL processes, $C$ and $S$, whose interaction may lock, and it builds (if possible) a BPEL process adapter $A$, which allows the two processes to successfully interoperate. The four main adaptation phases illustrated in Figure 4.7 are:

1. **Service Translation.** This phase is in charge of translating the BPEL descriptions of $C$ and $S$ into corresponding YAWL workflows. It inputs the BPEL processes of $C$ and $S$, and it outputs the workflows of $C$ and $S$. The translation is done using the BPEL2YAWL translator that will be described in Chapter 5. Note that, when plugged with the aggregation process described in Chapter 3, the translation of the two BPEL processes is to be done prior to the aggregation. In other words, this phase is necessary only when the client of the aggregation and adaptation methodology wishes to directly adapt two BPEL processes she provides as input. In order to provide a self-contained description of the adaptation process, we shall illustrate the translation of the two example BPEL processes (viz., ME and CC previously introduced in Subsection 4.2.1) into corresponding YAWL workflows.

2. **Adapter Generation.** This phase inputs the workflows of $C$ and $S$, and it outputs the workflow of an adapter $A$ (if possible). This phase makes the core of the behavioural adaptation and it builds the YAWL workflow of $A$ from the workflows of $C$ and $S$. It first generates the Service Execution Trees (SETs) of $C$ with respect to $S$ ($SET(C_S)$), as well as of $S$ with respect to $C$ ($SET(S_C)$), followed by the generation of the SETs of their duals ($SET(C_S)$ and $SET(S_C)$). Informally, when a service $X$ outputs a message $m$, a dual of $X$ is a service that inputs $m$, and vice versa. Next, $SET(A)$ is obtained by suitably merging $SET(C_S)$ and $SET(S_C)$. Finally, the YAWL workflow of $A$ is derived from $SET(A)$.

3. **Lock Analysis.** This phase verifies whether the YAWL-based aggregation of $C$, $A$, and $S$ locks. It inputs the three workflows, and should the aggregate have at least one lock-free traces, it outputs the workflow of $A$ (which we now call valid). The aggregation of the three workflows is to be done as described in Chapter 3. If the composition locks, we consider that the adaptation has
failed. Otherwise, we consider that the adaptation is successful. Note that the 
aggregate can be used for simulations in the YAWL engine, or for checking 
further properties using YAWL analysis tools.

4. **Adapter Deployment.** If the Lock Analysis phase is successful, the Adapter 
Deployment phase deploys the YAWL workflow of $A$ as a BPEL process, which 
can be used as a service-in-the-middle between $C$ and $S$. Hence, this phase 
inputs the YAWL workflow of $A$, and it outputs the BPEL process correspond-
ing to $A$. This phase is the inverse of the Service Translation phase, in that 
it employs a YAWL2BPEL translator. We shall describe the deployment of the 
adapter between the example ME and CC services.

**Service Translation**

As previously mentioned, in Chapter 5 we shall present a technique for translat-
ing BPEL processes into YAWL workflows. Its main strengths are that (1) it defines 
YAWL patterns for all BPEL activities, (2) it provides a compositional approach to 
construct structured patterns from suitably interconnecting other patterns, and (3) 
it handles events, faults and (explicit) compensation.
On the one hand, the pattern of each BPEL basic activity (with the exception of assign and compensate) is obtained by suitably instantiating the Basic Pattern Template (BPT). The BPT is a template of YAWL tasks, which serves both for identifying the translated activity (through an ActivitySpecificTask, or AST for short), as well as the control-logic of executing or skipping the activity. On the other hand, the pattern of each BPEL structured activity (together with assign and compensate) is obtained from the Structured Pattern Template (SPT) template. The SPT consists of a Begin (logically marking the initiation of the structured activity) and of an End pattern (logically marking the termination of the structured activity), as well as a pattern template (BPT or SPT) for each child activity. Each pattern inputs and outputs at most three types of control-flow links, called green, blue, and red lines. The green lines serve for translating the structural dependencies among BPEL activities. The blue lines are used for translating the BPEL synchronisation links, and the red lines are necessary for implementing the fault handling mechanism.

A BPEL process is translated into a YAWL workflow by instantiating the Process pattern. This leads to recursively instantiating the Begin(Process), FaultHandler, EventHandler (if any), and End(Process) patterns, as well as the BPT or SPT corresponding to the process activity. Note that instantiating a pattern takes into account the context in which the activity is placed inside the BPEL process. Namely, instantiating a pattern means adjusting the (number of) input and output lines, setting and mapping the inputs and outputs of the tasks in the pattern, as well as suitably interconnecting its child patterns.

For example, the YAWL workflows of the CC and ME services of our example can be seen in Figure 4.8. In the workflow of ME, the Begin(Process) and the End(Process) composite tasks, logically mark the initiation and the termination, respectively, of the BPEL process. The process activity, a sequence leads to generating the Begin(Sequence) as well as the End(Sequence) tasks. The first activity in the sequence is a receive, which gives the Receive composite task. Furthermore, the rest of the activities are translated correspondingly. (The numbers inside some of the task labels are used for disambiguation purposes only.) Please note the translation of the BPEL pick. The Begin(Pick) composite task contains the branch selection logic (basically a deferred choice construct [87]), and it outputs two tokens. One leads to executing the chosen branch, while the second leads to skipping the other branch (so as to achieve the dead-path-elimination).

The workflow of CC is built in a similar manner. However, the composite tasks representing the invoke ValidateLocation and invoke Explore activities output either “green” tokens, if the invocations succeed, or “red” tokens, if the invocations fail (i.e., faults are being raised). In the former case, the execution of the workflow

---

6Note that the two workflows are represented in a slightly simplified form with respect to the description given in Chapter 5 (e.g., the two synchronous invocations are represented as composite tasks instead of sequences Begin(Invoke) — End(Invoke), the default faultHandlers of the process and redundant GreenGates have not been represented, as well as the assign is represented in a compact form).
Figure 4.8: YAWL workflows corresponding to the CC and ME BPEL processes.

continues normally, and the green output of End(Sequence) leads to skipping the tasks inside the Begin(FaultHandler) → End(FaultHandler) zone (so as to achieve the dead-path-elimination). In the latter case, the execution of the faulty invocation is (immediately) followed by the execution of the tasks in the fault handling zone.

Please note that, in order not to add a burden to the aggregation and adaptation processes, this phase should be performed off-line. Consequently, we assume that service developers publish the contracts of $C$ and $S$ that define the service behaviour in terms of YAWL workflows.

**Adapter Generation**

The Adapter Generation phase consists of four steps which are discussed hereafter.

**Service Execution Trees**

This step automatically generates the Service Execution Trees (SETs) of the two services to be adapted. The SET of a BPEL process $X$ (or $SET(X)$ for short) is a tree describing all the possible scenarios of executing the basic activities (or
activities, for short) of $X$. Informally, the root of the SET is given by the activity (or activities) that can be executed first, while the leaves correspond to activities executed last. Each intermediary node represents the execution of one or more activities. A node consisting of more than one activity denotes a concurrent execution of the respective activities. Given a node $n$, child nodes of $n$ contain (distinct sets of) activities that can be executed immediately after executing the activities in $n$. Hence, one may think of each path in the tree as a service execution trace.

We generate $SET(X)$ through a reachability analysis of its corresponding YAWL workflow obtained during the Service Translation phase. For this purpose, we employ the reachability analysis previously described in Section 3.4. We recall that, in order to cope with loops in the process, our reachability analysis uses the modified reachability trees defined in [96]. Consequently, $SET(X)$ corresponds to the modified reachability tree (if the workflow is unbounded), or to the tree representation of the reachability graph (if the workflow is bounded). Furthermore, the generation of the execution traces described in Section 3.4 shows how to generate the conditions (viz., logical expressions) constraining the fulfilment of a service execution trace. As a result, each node of the SET can be labelled with a logical expression that states the condition in which the node activities are executed in the respective trace. Such conditions are due to the translation of BPEL switch activities, or due to synchronisation links. Note that the BPEL2YAWL translator allows us to cope – when adapting – both with synchronisation links and with the exceptional behaviour of BPEL.

The SET one obtains for a service $X$ contains all message exchanges of $X$ with other services. We call this the full-form of the SET, and, as previously indicated, we denote it by $SET(X)$. Similarly to the Service Translation phase, we argue that the designer of a service $X$ should provide the $SET(X)$ together with the contract of $X$ so as to lighten the processes of service composition and adaptation.

$SET(ME)$ is given in Figure 4.9(a). For example, the execution of the (synchronous) invoke ValidateLocation can be followed either by the invoke Explore, or by the invoke SubmitErr. The former is due to a successful execution of the invoke ValidateLocation activity, while the latter is executed in the case of a fault being received by the invoke ValidateLocation. Furthermore, the successful termination of the sequence activity of the BPEL process leads to the dead-path-elimination being employed inside the pattern implementing the faultHandler of the BPEL process. This is indicated in $SET(ME)$ by the dark coloured invoke SubmitErr and rev Logout nodes.

From (the full-form of) $SET(X)$ we derive next the (compact-form of) $SET$ of $X$ with respect to another service $Y$, with which $X$ interacts. We denote it by $SET(X_Y)$. Informally, from the original $SET(X)$ we only keep message exchanges between $X$ and $Y$.

First, all message exchanges (viz., receive/reply/invoke) of $X$ with services other than $Y$, as well as all other basic activities (e.g., assign), and all skipped activities are set to empty activities. We denote the resulting SET as $SET(X_Y')$. For exam-
4.2. BEHAVIOURAL SERVICE ADAPTATION

Figure 4.9: (a) $SET(ME)$, (b) $SET(ME^*_{CC})$, (c) $SET(ME_{CC})$, and (d) $SET(CC_{ME})$.

ple, the invoke $ValidateLocation$ and the invoke $Explore$, which $ME$ performs on the $Logger$ Service and $Explorer$ Service, respectively, are set to $empty$ when computing $SET(ME^*_{CC})$. The result of applying this transformation on $SET(ME)$ (i.e., $SET(ME^*_{CC})$) is given in Figure 4.9(b).

Second, each $empty$ node in $SET(X^*_Y)$ (with the exception of the root) is removed from the tree, and its sub-trees (if any) are merged with its parent nodes. What one obtains is $SET(X_Y)$. Note that the merge process applied at a node $N$ of $SET(X^*_Y)$ also removes duplicate subtrees of $N$. For example, by removing the three empty nodes of $SET(ME^*_{CC})$ (Figure 4.9(b)), we get two identical subtrees ($invoke SubmitErr$ → $receive Logout$) at node $receive SetCoords$. The merge at $receive SetCoords$ will then remove one duplicate. $SET(ME_{CC})$ is represented in Figure 4.9(c). Furthermore, the construction of $SET(CC_{ME})$ is similar, and we present it in Figure 4.9(d).

Dual SETs

This step generates for each service $X$ (to be adapted), the SET of a dual of $X$ with respect to another service $Y$. Basically, when $X$ receives a message $m$ from $Y$, a
dual of $X$ with respect to $Y$ (denoted by $\text{SET}(\overline{X}_Y)$) acts somewhat “as $Y$ should” and sends a message $m$ to $X$, and vice versa. One obtains the $\text{SET}(X_Y)$ from the $\text{SET}(X_Y)$ by replacing asynchronous invokes with receives (and vice versa), and synchronous invokes\footnote{Viz., pairs $\text{Begin}(\text{Invoke}) \rightarrow \text{End}(\text{Invoke})$.} with pairs $\text{receive} \rightarrow \text{reply}$ (and vice versa). $\text{SET}(\overline{ME}_CC)$ and $\text{SET}(\overline{CC}_{ME})$ are depicted in Figure 4.10(a) and (b), respectively.

**Adapter SET**

The $\text{SET}$ of an adapter $A$ ($\text{SET}(A)$) mediating the interaction of two services, $C$ and $S$, is obtained by suitably merging $\text{SET}(\overline{C}_S)$ with $\text{SET}(\overline{S}_C)$. This process consists of two steps, as follows.

During the first step, we match activities of $\text{SET}(\overline{C}_S)$ with activities of $\text{SET}(\overline{S}_C)$ with the following two rules:

- An asynchronous invoke $Op$ of $\text{SET}(\overline{C}_S)$ matches a receive $Op$ of $\text{SET}(\overline{S}_C)$, and vice versa, and
4.2. BEHAVIOURAL SERVICE ADAPTATION

4.2.1. A synchronous **invoke Op**\(^8\) of \(SET(C_S)\) matches a pair **receive Op \(\rightarrow\) reply Op** of \(SET(S_C)\), and vice versa.

Then, we express each match as a **data-flow dependency** (or dependency, for short), which emerges at the **receive** and targets the **invoke**, in the case of an asynchronous message exchange, or as a pair of dependencies, one emerging at the **receive** and targeting the **invoke**\(^9\), and another one emerging at the **invoke**\(^10\) and targeting the **reply**, in the case of synchronous message exchanges.

We call an activity that is target of at least one dependency as “constrained”. Otherwise, we say that the activity is “unconstrained” (with respect to the data-flow dependencies between the two SETs). For example, **invoke Login** and **receive JobId** of \(SET(ME_{CC})\) match **receive Login** and **invoke JobId**, respectively, of \(SET(CC_{ME})\). (See Figure 4.10(a) and (b).) Informally, a dependency indicates that the adapter has to wait first for a message from one of the two services, and then (possibly at a later moment) it forwards it to the other service. In other words, a dependency from \(X\) to \(Y\) says that the adapter has to execute \(X\) before executing \(Y\). Note that the interpretation in the case of multiple dependencies emerging from different activities \(X_k\) and targeting an activity \(Y\), is that for the execution of \(Y\) it suffices to execute only one activity \(X_k\). This is the case of the **invoke Logout** (1) and **invoke Logout** (2) of \(SET(ME_{CC})\).

As earlier mentioned, each path in \(SET(X)\) is an execution trace of \(X\). During the second step, we compute the merge of all possible pairs of traces \((\vec{c}, \vec{s})\), where \(\vec{c} = <c_1, c_2, \ldots, c_m>\) is a trace of \(SET(C_S)\), and \(\vec{s} = <s_1, s_2, \ldots, s_m>\) is a trace of \(SET(S_C)\). Such a merge can lead either to a **success**, or to a **failure**. In the former case, the merge of \(\vec{c}\) and \(\vec{s}\) gives a (successful) trace \(a\) of the adapter \(A\) (and consequently a path in \(SET(A)\)).

At each step, the merge process compares nodes \(\vec{c}_i\) and \(\vec{s}_j\), by starting from the roots of the two traces, and it produces a node \(a_k\). In terms of BPEL activities, one may think of the node \(a_k\) as a **sequence** containing a **flow**. The merge algorithm basically adds activities of the two nodes \((\vec{c}_i\) and \(\vec{s}_j\)) either inside the **flow**, or inside the **sequence**, yet following the **flow**. For simplicity, we informally describe hereafter the algorithm of merging two nodes containing each one activity only, and each being the target of at most one dependency. (The general case of merging nodes with multiple activities and multiple constraints is analogous.)

1. If \(\vec{c}_i\) is unconstrained, then add \(\vec{c}_i\) to the **flow** inside \(a_k\) (e.g., merging **receive JobId** and **receive SetCoords**). Please note that in the case of an unconstrained **invoke** activity, the merge process (of the two traces) returns with a **failure**. We do so in order to avoid the generation of (arbitrary) messages by the adapter.

---

\(^{8}\)Viz., a pair \(\text{Begin(Invoke Op)} \rightarrow \text{End(Invoke Op)}\).

\(^{9}\)Viz., the **Begin(Invoke)** pattern that marks the start of the synchronous invocation.

\(^{10}\)Viz., the **End(Invoke)** pattern that marks the termination of the synchronous invocation.
2. Otherwise, if \( c_i \) is constrained by \( \overline{s_j} \) such that \( J < j \) (i.e., from the point of view of executing the trace \( \overline{s} \), the activity of \( \overline{s_j} \) has already been executed), then add \( c_i \) to the flow (e.g., merging \textit{invoke SetCoords} and \textit{invoke SubmitRep}).

3. Otherwise, if \( c_i \) is constrained by \( \overline{s_j} \) such that \( J = j \) (i.e., the activity of \( \overline{s_j} \) is ready to be executed), then add \( c_i \) to the sequence, following the flow (e.g., merging \textit{invoke Login} and \textit{receive Login}).

4. Otherwise, if \( c_i \) is constrained by \( \overline{s_j} \) such that \( j < J \) (i.e., the activity of \( \overline{s_j} \) is not executable yet), then we say that the trace \( \overline{s} \) is “stalled” (e.g., assume merging \textit{invoke SetJob} and \textit{receive SetCoords}).

Next, the algorithm repeats 1.–4. for \( \overline{s_j} \). For example, one may see in Figure 4.10(c) the result of merging the roots of \( ME_{CC} \) and \( CC_{ME} \). (The elimination of the flow is due to the fact that it contains one activity only.)

If both traces are stalled, then we have a lock between the two traces, and hence a \textit{failure} in merging the two traces. Otherwise, the algorithm continues by comparing the node \( c_i \) (if \( \overline{s} \) is stalled) or \( c_{i+1} \) (if \( \overline{s} \) is not stalled) with the node \( \overline{s_j} \) (if \( \overline{s} \) is stalled) or \( \overline{s}_{j+1} \) (if \( \overline{s} \) is not stalled). If the merge has added to the trace \( a \) all nodes of one of the two traces \( \overline{s}/\overline{a} \), it simply appends at the end of \( a \) the remaining sequence of nodes of the other trace \( \overline{s}/\overline{a} \). If all nodes of both \( \overline{s} \) and \( \overline{a} \) have been added to \( a \), then we have a \textit{success}, and \( a \) represents a (successful) trace of the adapter \( A \).

Next, we derive \( SET(A) \) by merging all successful traces \( a \) of \( A \). If no such successful traces exist, then the algorithm generating the adapter fails, as the mismatches between the two interacting processes cannot be solved. For example, if the root of \( SET(\overline{UC}) \) consists of an \textit{invoke Op}_1 and if the root of \( SET(\overline{SC}) \) consists of another \textit{invoke Op}_2, then we have a deadlock as each service is waiting to receive a message from the other. Consider a set \( \{a^1, a^2, \ldots, a^p\} \) of successful adapter traces. The merge algorithm, in this case, starts by considering \( SET(A) \) to be \( a^1 \). Then for all nodes \( a^k_i \) of the other traces \( a^k \), it checks whether \( a^k_i \) is contained in \( SET(A) \) at depth \( i \). If so, it marks the respective position in the tree, and it chooses the next node in the sequence (i.e., \( a^k_{i+1} \)). Otherwise, it adds the rest of the trace \( a^k \), including the node \( a^k_i \), as a branch splitting from the last marked node in \( SET(A) \).

For our example we get only two successful traces of the adapter. The first one, denoted by (5) in Figure 4.10(d) is obtained by merging the traces denoted by (1) and (3) of \( SET(ME_{CC}) \) and of \( SET(CC_{ME}) \), respectively, while the second one, denoted by (6) is obtained by merging traces denoted by (2) and by (4). These two adapter traces are then merged into the adapter seen in Figure 4.10(d).

**Adapter Workflow**

If the adapter has at least one successful trace, then the adaptation process generates next the YAWL workflow of the adapter \( A \) from \( SET(A) \) as described hereafter.
(Note that the process of constructing the workflow of A uses the translation patterns defined in Chapter 5.)

Initially, it generates the Begin(Process) and the Begin(Sequence), as well as the End(Sequence) and the End(Process) patterns, which logically mark the initiation of the business process and of its activity, as well as their termination, respectively. The former two, as well as the last two are to be linked in a sequence. (See Figure 4.11.) Basically, generating the pattern of a basic activity simply consists of instantiating the Basic Pattern Template (e.g., setting the name, inputs, and outputs of its ActivitySpecificTask), while generating the pattern of a structured one reduces to instantiating its Begin and its End pattern, as well as the pattern of each child activity.

For each node $n$ in $SET(A)$, starting with its root, the algorithm generates and adds to the workflow the pattern(s) corresponding to the activity (activities) contained in $n$. If $n$ consists of one activity only, then the pattern of its (basic) activity is produced and suitably linked in the workflow as output of the pattern corresponding to the parent node of $n$ (or to Begin(Sequence) if $n$ is the root). For example, the receive Login root of $SET(A)$ leads to a Receive pattern being generated and linked as output of Begin(Sequence). Otherwise, if $n$ consists of multiple activities, then the pattern given by the node is a Flow, which includes the patterns of each activity in the node.

Next, if $n$ has one child node only, the adaptation process continues with its child. Else, if $n$ has more than one successor, then we have three possibilities:

1. If all child nodes of $n$ contain each one receive only, and if there are no conditions constraining their execution\footnote{We recall that such conditions are due to the translation of the BPEL switch activity or to synchronisation links.} then the resulting pattern is a Pick having the respective receives as onMessage tasks in Begin(Pick), and for each branch is generated a Sequence pattern. The generation process continues then on each subtree having as root a child of $n$ (excluding the child of $n$ already considered as onMessage inside the Pick). Else,

2. If all child nodes of $n$ are constrained by (disjoint) conditions, then a Switch pattern is produced with the respective conditions as guards, and for each branch of the Switch, a Sequence pattern is generated. The algorithm continues next on each branch of the subtree with the root $n$. Else,

3. In all other cases, the adaptation process aborts, as the adapter cannot be successfully constructed due to a non-deterministic (other than pick) behaviour. For example, if $n$ has two unconstrained children, one invoke $Op_1$ and one receive $Op_2$, then the adapter cannot “know” whether it should wait for a message, or whether it should send a message.

The YAWL adapter one obtains for our example is presented in Figure 4.11.
Figure 4.11: YAWL workflow of an adapter for CC and ME.

Lock Analysis

This phase of the adaptation process is concerned with the lock-freedom of the interaction between the adapter $A$, and the two services, $C$ and $S$. In a first step, the core aggregation process (described in Subsection 3.4.4) suitably builds the composition of $C$, $A$, and $S$. In short, the aggregation of $C$, $A$, and $S$ has the following particularities:

- Data exchanged by $C$ and $S$ flows only through the adapter $A$. In other words, the aggregation process does not (directly) link tasks belonging to $C$ and $S$. This is achieved as follows. Each data-flow dependency emerging at activity $X$ of $SET(CS)$ and targeting another activity $Y$ of $SET(SC)$ (Figure 4.10 (a) and (b)) generates (at most) two data-flow dependencies in the aggregated workflow:

1. A dependency emerging at task $X$ of $C$ and targeting task $\overline{X}$ of the adapter workflow $A$, and
2. Another dependency emerging at task $\overline{Y}$ of $A$ and targeting task $Y$ of $S$.

Note that, $\overline{X}$ could be missing from $SET(A)$ if all the traces of $SET(\overline{CS})$ that contain $\overline{X}$ lead to merge failures with the traces of $SET(\overline{SC})$. In this case, $A$ will not have an $\overline{X}$ task, and furthermore, in the aggregation of $S$ with $A$ and $C$ the task $X$ will have an output data-flow enabler without outgoing links. In other words, $A$ does not have a task $\overline{X}$ that could input the output message of task $X$ of $C$. The scenario is dual for $\overline{Y}$. In this case, $A$ does not have a task $\overline{Y}$ that could output the message requested by the task $Y$ of $S$. 
Basically, the two dependencies constrain \( S \) from executing \( Y \) before receiving the outputs of \( X \) from \( A \), which previously received them from \( C \).\(^{12}\) For example, the *Invoke Login* pattern of the CC workflow is to be connected as input of the *Receive Login* pattern of the adapter workflow, while the *Invoke Login* pattern of the adapter is to be connected as input of the *Receive Login* pattern of the ME. In this way, the adapter first receives the login information from the CC, and then (at a later moment) it forwards it to the ME. (The scenario is similar for dependencies emerging at activities \( \overline{Y} \) of \( SET(S_C) \) and targeting activities \( \overline{X} \) of \( SET(S_S) \).)

- Furthermore, each such dependency from a task \( X \) to another task \( Y \) is expressed in the aggregate as a blue (synchronisation) link\(^{13}\) that emerges at the pattern of \( X \), and that targets the pattern of \( Y \).

Note that, if \( X \) (or \( Y \)) is an *onMessage* task of a *Begin(Pick)* pattern, then the *Begin(Pick)* pattern should be replaced by its internal workflow net, and moreover, the *BeginPick* composite task of *Begin(Pick)* should be substituted by its inner workflow net as well. The reason for doing so is that YAWL workflow links cannot cross the boundaries of composite tasks. For example, this is the case for the data-flow dependency between the *invoke SubmitReport* task of \( CC \) and the *onMessage SubmitReport* task of the adapter.

- Finally, due to the specific form of the patterns translating BPEL activities\(^{14}\), and due to the fact that we model the data-flow of the aggregate through blue synchronisation links, the Task Expansion step of the aggregation does not need to explicitly separate the control- from the data-flow.

The second step performs a reachability analysis of the composite (as described in Subsection 3.4.1) so as to check whether it locks (e.g., a non-final node of the reachability graph that has no outgoing links corresponds to a deadlock). The output of this step is one of the following:

- If all traces of the aggregate are lock-free, then \( A \) is a *full adapter* for \( C \) and \( S \). Otherwise,

- If some (yet not all) of the traces of the aggregate are lock-free, then \( A \) is a *partial adapter*, as there are interaction scenarios that cannot be resolved.

\(^{12}\)Note that, the dependency between \( \overline{X} \) and \( \overline{Y} \) already “contributed” to the generation of the workflow of \( A \) (it was fused in the control-flow of \( A \)).

\(^{13}\)We recall that blue links in the workflow correspond to BPEL synchronisation links. These synchronisation links in the aggregate employ a *true transitionCondition* in order to output a blue token whenever the \( X \) pattern executes successfully.

\(^{14}\)The YAWL patterns translating BPEL activities are roughly composite tasks with OR-splits, and they employ *BlueGate* tasks with AND-joins to wait for the statuses of the incoming blue links. (See Chapter 5 for details on the patterns.)
• If the aggregate does not have lock-free traces, then we consider that the adaptation has failed.

It is important to note that, together with the aggregation of \( A \) with \( S \) and \( C \), this phase also constructs the composition of \( A \) with \( S \). While the former serves for validating the adapter, the latter (viz., the adapted service) will make the output of the adaptation process, together with the adapter \( A \).

In order to ease the presentation of the methodology we do not illustrate the aggregation of the \( CC \) and \( ME \) with the adapter previously generated. However, note that the adapter given in Figure 4.11 is a full adapter that copes with the mismatches between \( CC \) and \( ME \) as follows:

1. It receives in parallel the coordinates form the \( CC \) and the job id from the \( ME \),
2. It first invokes \( SetJob \) and then \( SetCoords \) of the \( ME \), and
3. It forwards the job id to the \( CC \) only when it receives a report from the \( ME \).

**Adapter Deployment**

If the Lock Analysis phase has validated \( A \) as a full/partial adapter, then the Adapter Deployment phase generates the BPEL processes of both, the adapter \( A \), and the composition between \( A \) and \( S \), from their YAWL workflows obtained during the previous phase. The deployment process (viz., the \texttt{YAWL2BPEL} translator) works by parsing the YAWL workflow with respect to the patterns defined in the \texttt{BPEL2YAWL} translator (described in Chapter 5). For example, the \texttt{Pick} pattern in Figure 4.11 leads to the generation of a BPEL \texttt{pick} activity with two branches guarded by \texttt{onMessage SubmitRep}, and \texttt{onMessage SubmitErr}, and each branch activity is a \texttt{sequence}.

Please note that, although not explicitly represented in the pictures, the YAWL patterns translating the BPEL activities contain all the necessary information for the inverse, \texttt{YAWL2BPEL} translator (e.g., \texttt{partnerLink}, \texttt{portType}, \texttt{operation}, and \texttt{variable} attributes in the case of a \texttt{receive}, and so on). From the YAWL workflow of the adapter in Figure 4.11, one obtains the following BPEL (adapter) process:

```xml
<process name="Adapter_for_CC_and_ME"><sequence>
  <receive op="Login" from CommandCentre var="loginInfo"/>
  <invoke op="Login" of MarsExplorer var="loginInfo"/>
  <flow>
    <receive op="SetCoords" from CommandCentre var="coords"/>
    <receive op="JobID" from MarsExplorer var="id"/></flow>
  <receive op="SetJob" from CommandCentre var="jobDetails"/>
  <invoke op="SetJob" of MarsExplorer var="jobDetails"/>
  <invoke op="SetCoords" of MarsExplorer var="coords"/>
  <pick>
    <onMsg op="SubmitRep" from MarsExplorer var="report">...
```
4.2. BEHAVIOURAL SERVICE ADAPTATION

4.2.3 Informal Proof of Correctness

In the following we informally prove the correctness of the core behavioural adaptation technique. The core of the behavioural adaptation inputs two workflows $S$ and $C$ corresponding to two BPEL processes that present interaction mismatches (roughly due to a different ordering of the operations), and it outputs the workflow of an adapter $A$ (if any) such that $S$, $A$, and $C$ successfully interact. Recall that the Contract Validation employs a reachability analysis of the aggregation $SAC$ of the three workflows so as to verify whether $A$ solves the interaction mismatches of $S$ and $C$. Informally, we say that $A$ solves the interaction mismatches of $S$ and $C$ if there exists at least one trace $T_{SAC}$ of the aggregate such that:

1. for any task $t_{rec}^S := Receive_C(m)$ (viz., “receive message $m$ from $C$”\(^{15}\)) of $S$ executed in $T_{SAC}$, there exists a prior execution of a task $t_{inv}^C := Invoke_S(m)$ (viz., “send message $m$ to $S$”) of $C$ in $T_{SAC}$, and vice versa if we replace $S$ and $C$ — for asynchronous operations that input and output messages.

2. for any task $t_{end}^S := End(Invoke)_C(out)$ (viz., “receive message $out$ from $C$”) of $S$ executed in $T_{SAC}$, $T_{SAC}$ previously executed first a $t_{beg}^S := Begin(Invoke)_C(in)$ (viz., “send message $in$ to $C$”) of $S$, then a $t_{rec}^C := Receive_S(in)$ (viz., “receive message $in$ from $S$”) of $C$, and finally a $t_{rep}^C := Reply_S(out)$ (viz., “send message $out$ to $S$”) of $C$, and vice versa if we replace $S$ and $C$ — for synchronous operations that input $in$ and output $out$.\(^{16}\)

In the following we only prove 1. since 2. has a similar proof.

**Proof.** Once again, the proof is by construction. First, note that $A$ is in charge of passing messages between $S$ and $C$, and it does not attempt to solve the interaction mismatches of $S$ and $C$ with other parties. Consequently, the behavioural adaptation only “sees” the data-flow due to messages exchanged between

\(^{15}\)In order to ease the notation we do not specify the operation of $S$ that inputs the message $m$.

\(^{16}\)Roughly, we represent synchronous BPEL invocations in YAWL as a sequence of patterns $Begin(Invoke) \rightarrow End(Invoke)$ (see Chapter 5 for details). However, since the two synchronous invocations of the Mars Explorer process (viz., $invoke Validate Location$ and $invoke Explore$) do not represent message exchanges with the Command Centre, we represent them in Figure 4.8 in a simplified form as composite tasks.
Since \( t_A \) is an immediate consequence of the way in which \( t_A \) is executed in a successful trace \( T_{SAC} \) of the aggregate workflow. This implies that there exists (at least one) task \( t_s^{\text{inv}} := \text{Invoke}_S(m) \) of \( S \). Otherwise, the generation of \( A \) would have failed because \( \text{SET}(\overline{S_C}) \) would have contained a unconstrained \( \text{Invoke}_S(m) \) node as the dual of \( t_s^{\text{rec}} \) (see the generation of \( \text{SET}(A) \) in Subsection 4.2.2).

Furthermore, the execution of \( t_s^{\text{rec}} \) in \( T_{SAC} \) implies a prior execution of a task \( t_A^{\text{inv}} := \text{Invoke}_S(m) \) of \( A \). The existence of (at least one) \( t_A^{\text{inv}} \) is guaranteed by the fact that, otherwise, \( t_s^{\text{rec}} \) would have been constrained by \( \text{SAC} \) a disconnected input data enabler dummy because \( A \) would not have a task that could produce the message \( m \) requested by \( t_s^{\text{rec}} \). Since \( t_s^{\text{rec}} \) is executed in \( T_{SAC} \), and because its execution is constrained by the execution of a \( t_A^{\text{inv}} \) task of \( A \) (because there exists a link emerging at (the output data enabler of) \( t_A^{\text{inv}} \), and targeting (the input data enabler of) \( t_s^{\text{rec}} \)), we get that at least one \( t_A^{\text{inv}} \) is executed in \( T_{SAC} \) before \( t_s^{\text{rec}} \).

The existence of \( t_A^{\text{inv}} \) in \( A \) implies the existence of a \( t_A^{\text{inv}} \) node in \( \text{SET}(A) \). This is an immediate consequence of the way in which \( A \) is produced from \( \text{SET}(A) \). Since \( t_A^{\text{inv}} \) also belongs to \( \text{SET}(\overline{S_C}) \) (by construction of \( \text{SET}(\overline{S_C}) \)) and because it is constrained by (at least one) \( \text{Receive}_C(m) \) in \( \text{SET}(\overline{C_S}) \) (otherwise the generation of \( A \) would have failed; see above) we obtain that each trace of \( SET(A) \) that contains \( t_A^{\text{inv}} \) also contains an ancestor node \( t_A^{\text{rec}} := \text{Receive}_C(m) \), and consequently there is (at least one) \( t_A^{\text{rec}} \) task in \( A \). This immediately results from the algorithm of merging the traces of \( \text{SET}(\overline{S_C}) \) and \( \text{SET}(\overline{C_S}) \) into \( \text{SET}(A) \), which cannot consider \( t_A^{\text{inv}} \) in a trace of \( A \) if it did not already consider a \( t_A^{\text{rec}} \). Because each \( t_A^{\text{inv}} \) node of \( \text{SET}(A) \) has an ancestor node \( t_A^{\text{rec}} \), we obtain that there exists an ancestor (possibly structured) pattern of \( t_A^{\text{inv}} \) (viz., \( \text{Ancestor}(t_A^{\text{inv}}) \)) in \( A \) such that the \( t_A^{\text{rec}} \) task and the \( \text{Ancestor}(t_A^{\text{inv}}) \) pattern are linked sequentially in \( A \) (viz., \( \text{Begin}(\text{Sequence}) \rightarrow \ldots \rightarrow t_A^{\text{rec}} \rightarrow \text{Ancestor}(t_A^{\text{inv}}) \rightarrow \ldots \rightarrow \text{End}(\text{Sequence}) \)). This is again immediate from the transformation of \( \text{SET}(A) \) into the workflow of \( A \).

Since \( t_A^{\text{inv}} \) is executed in \( T_{SAC} \) we obtain that the (beginning of the) \( \text{Ancestor}(t_A^{\text{inv}}) \) pattern was also executed in \( T_{SAC} \) before \( t_A^{\text{inv}} \), and consequently that \( t_A^{\text{rec}} \) was executed in \( T_{SAC} \) before the (initiation of) \( \text{Ancestor}(t_A^{\text{inv}}) \), hence before the execution of \( t_A^{\text{inv}} \). Furthermore, the execution of \( t_A^{\text{rec}} \) in \( T_{SAC} \) implies the prior execution of a \( t_C^{\text{inv}} \) task in \( T_{SAC} \) because there exists (at least one) data-flow dependency link from tasks \( t_C^{\text{inv}} \) to \( t_A^{\text{rec}} \) in \( \text{SAC} \). Consequently, we obtain that \( T_{SAC} \) executed \( t_C^{\text{inv}} \), followed at a later moment by \( t_A^{\text{rec}} \), then \( t_A^{\text{inv}} \), and finally \( t_s^{\text{rec}} \), that is, prior to the execution of \( \text{Receive}_C(m) \) of \( S \) in \( T_{SAC} \) there exists an execution of \( \text{Invoke}_S(m) \) of \( C \).
4.3 Complexity Analysis

In the following we informally discuss the complexities of the functional and behavioural adaptation techniques.

- **Functional Adaptation.** Briefly, given a functional query $Q$ that requests services with certain IOs, the functional adaptation process generates starting from matched services $S$ adapted services $S'$ whose behaviour is compliant with the request $Q$ (whenever possible). Informally, the workflow of a service $S'$ corresponds to a refined behaviour of the initial service $S$ that enforces the needed adaptation.

  - **Reachability Analysis and Trace Table Generation.** The reachability analysis and the generation of the trace table are as defined in Subsection 3.4.1 and 3.4.2, respectively. Consequently, the generation of the trace table is EXPSPACE-hard.

  - **Service Matching.** In Section 3.7 we showed that the generation of the matchmaker graph takes $O(2^N)$ steps, where $O(N)$ is the total number of traces in the registry.

  - **Contract Generation.** Hereafter we compute the complexity of adapting the workflow of one matched service ($S$). Assume $S$ has $n$ traces, and $m$ tasks. Then, the inspection of $S$ for redundant tasks takes $O(n \times m)$ steps. We can have at most $m$ redundant tasks, hence their elimination from $S$ takes $O(m)$ steps. Consequently, the contract generation phase takes $O(n \times m)$ steps for the computation of an adapted contract $S'$.

- **Behavioural Adaptation.** In short, given two services (e.g., BPEL processes) $S$ and $C$, the behavioural adaptation technique attempts to generate an adapter (e.g., BPEL process) $A$ that can be used as a service-in-the-middle between $S$ and $C$ as a way to overcome their interaction mismatches.

  - **Service Translation.** Assume $S$ has $n_S$ activities and $C$ has $n_C$ activities. The translation of the two services takes $O(c \times (n_S + n_C))$ steps, or $O(n_S)$, if we assume that $n_S > n_C$, and that the translation does not produce more than $c$ tasks for each activity.

  - **Adapter Generation.** The complexity of the adapter generation phase sums the following. The generation of $SET(S)$ (and similarly $SET(C)$) is EXPSPACE-hard, since the process is similar to the reachability analysis and trace table generation employed by the aggregation process. Assume now that $S$ has $n_S$ traces and $m_S$ tasks. The generation of $SET(S_C)$ takes $O(n_S \times n_C)$ steps, and so does the generation of $SET(S_C)$. Dually, the efforts of generating $SET(C_S)$ and $SET(C_S)$ take each $O(n_C \times n_S)$. The generation of $SET(A)$ can be computed as the sum between the effort of
matching the nodes of \( \text{SET}(\overline{S_C}) \) and \( \text{SET}(\overline{C_S}) \), the effort of merging their traces into the adapter traces, and the effort of generating \( \text{SET}(A) \) from the traces of the adapter. The matching takes \( O((n_S \cdot m_S) \cdot (n_C \cdot m_C)) \) steps, or \( O((n_S \cdot m_S)^2) \), with the assumption that \( n_S > n_C \), and \( m_S > m_C \). Furthermore, the merge step takes \( O((n_S \cdot n_C) \cdot (m_S + m_C)) \) steps, that is, \( O(n_S^2 \cdot m_S) \). Finally, the complexity of merging the adapter traces into \( \text{SET}(A) \) is also \( O(n_S^2 \cdot m_S) \), since the adapter has at most \( n_S \cdot n_C \) traces, and each adapter trace has at most \( m_S + m_C \) nodes. Hence, generating \( \text{SET}(A) \) takes \( O(n_S^2 \cdot m_S) \) steps. Finally, the process of deriving the adapter workflow from \( \text{SET}(A) \) is linear with respect to the size of \( \text{SET}(A) \), and hence it takes \( O(n_S^2 \cdot m_S) \) steps.

- **Lock Analysis.** The lock analysis phase first aggregates \( S \), \( A \), and \( C \) and then it employs a reachability analysis of the aggregate so as to determine whether \( A \) successfully mediates the interaction of \( S \) and \( C \). Both steps of the lock analysis phase are similar to the core aggregation and contract generation and respectively, contract validation phases of the aggregation process described in Chapter 3. Hence, the lock analysis phase is EXPSPACE-hard.

- **Contract Deployment.** The last phase of the behavioural adaptation process is the deployment of the adapter workflow as a BPEL process. Dually to the service translation phase, the translation is linear with respect to the size of the workflow of \( A \), and hence it takes \( O(n_S^2 \cdot m_S) \) steps, since the number of tasks of \( A \) is linear with respect to the size of \( \text{SET}(A) \).

## 4.4 Related Work

Web service adaptation is in its early stages and current approaches feature only partial solutions to the issues of adaptation. Hau et al. [40] present a service adaptation framework, called The ICENI Semantic Service Adaptation Framework. Their approach basically features a semantic matchmaking as well as a service adaptation process. Services are described in terms of OWL in order to allow to be semantically matched based on their ontology annotations. In a first phase, the adaptation process finds services that are conceptually equivalent to the request. The semantic matchmaking uses two types of inference operations – “class” and “property” inferences. Class inferences are used to select services conceptually equivalent to the client interface method, while property inferences are used to filter these services by inspecting properties and their relations. In a second phase, their engine analyses the information gathered during matchmaking by using graph transformation rules. In case of a match, the result is an architecture independent binding that provides to the client the required interface. As the authors note, the aim of the adaptation service is to find a transformation function with the required method
signature as domain and the list of conceptually compatible service signatures as the range. Hence, their approach deals with signature mismatches only and not with behavioural ones.

Syu [82] proposes an ontology-based approach to the automated adaptation of Web services. The adaptation system makes use of ontology languages such as OWL and OWL-S to tackle three simple cases of adaptation – permutation, modification, and combination – of input parameters. Basically, the approach treats the adaptation of one requested parameter to one and respectively, two corresponding parameters in the target operation, and vice versa. However, Syu argues that one can easily extend the proposed architecture by simply adding compatible OWL rules.

Iyer et al. [42] present a methodology for Web service adaptation through an adaptation architecture. They argue that Web services are the glue that will link together Web deployed components to form Web applications. In other words, they show how to achieve interoperability between different SOAP Web services in a platform independent manner, with the help of XML scripts and XSL. Although their approach does not require the programming of custom adapters (i.e., made of other adapters), their generation is manual. Moreover, they address adaptation at a signature level only, resembling early attempts to component adaptation.

Kaykova et al. [46] describe a general approach for the adaptation of heterogeneous industrial resources into an unified environment. Their focus is on performing semantic adaptation, and they show how to semantically adapt such resources by using a two step transformation for an use case. Yet, their approach – as [40] – relies on black-box views of services and on semantically annotated signatures.

Ponnekanti and Fox [78] propose a framework for coping with structural, value, encoding, and semantic incompatibilities among services. However, their approach – as [40, 42, 82] – relies on black-box views of services.

Brogi et al. [14] present a process algebra (CCS) based formalisation of WSCI. As the authors note, some of the benefits of such formalisation are the definition of compatibility and replaceability tests between Web services. The paper describes a technique for checking whether communicating Web services can interoperate successfully, or whether the concrete specification of adapters that mediate their interaction can be automatically generated. Still, a downside of their approach is that the high-level adapter specification has to be manually generated.

Benatallah et al. [10] describe an approach for the generation of replaceability adapters based on mismatch patterns. In short, the approach provides a classification of interface and protocol differences as mismatch patterns, as well as corresponding adapter templates for tackling such mismatches. The adapter templates basically consist of code or pseudo-code that describe the implementation of adapters that can resolve the differences captured by the patterns. However, their approach does not describe how to capture complex protocol mismatches (through pattern compositions). Furthermore, given two protocols to be adapted, the designer has to identify the mismatch(es) between the two, and she is in charge of generating the
adapter code (e.g., the provision of the template parameters).

Kongdenfha et al. [50] present a technique for the runtime adaptation of BPEL business process specifications so that they become compliant with external client specifications. Roughly, the adaptation consists in running snippets of BPEL code at particular moments during the process’ execution (called adaptation templates) in order to solve various types of mismatches (e.g., different operation names, parameter types, or order of messages). Similarly to [10], their approach does not address the composition of the adapter templates for solving (complex) behavioural mismatches between (complex) BPEL processes involving e.g., faulty or exceptional behaviour. Furthermore, the developer has to identify the activities where adaptation is needed, and she has to provide the adaptation parameters.

To the best of our knowledge, our adaptation process is the first to take into account both semantics and behaviour information, and to tackle both functional and behavioural client queries. Furthermore, it straightforwardly integrates with the aggregation process into a methodology for the aggregation, adaptation, and verification of service contracts. Last but not least, we argue that the adaptation can be successfully employed to automatically generate (full/partial) BPEL adapters for BPEL processes presenting interaction mismatches.

4.5 Discussion

In this Section we illustrated a service adaptation process that features, on the one hand, a functional adaptation in the form of customising service behaviour to functional client requests, and on the other hand, a behavioural adaptation that generates service adapters for communicating services that present mismatches in their interaction protocols.

We would like to stress out the fact that the adaptation process can be either plugged with the aggregation process for locating and enforcing (compositions of) services to meet client’s needs, or it can be employed by itself, for the generation of adapters and/or adapted services if the client provides the service to be adapted together with the query.

The core of the adaptation takes as input a (composite) service contract \( S \) and a client query \( Q \), and it deals with adapting \( S \) so as to satisfy \( Q \). In short, the core of the functional adaptation process, based on an analysis of the execution traces of \( S \) and \( Q \), refines the behaviour of \( S \) by removing parts of its workflow that are redundant with respect to the query, and then by suitably re-linking the remaining workflow tasks. The resulting service contract \( S' \) (if any) offers the same functional description as \( Q \) (viz., the black-box views of \( S' \) and \( Q \) are the same). We recall that the methodology can also be used to generate services \( S' \) that are compositionally compatible with a functional request \( Q \), by providing as input a dual of \( Q \), which is a black-box that inputs the outputs of \( Q \) and that outputs the inputs of \( Q \). The core of the behavioural adaptation process builds from the service execution
trees of $S$ and $Q$ the execution trees of their duals with respect to each other, viz., $SET(\overline{S})$ and $SET(\overline{Q})$, respectively. Then, from these adapter-views of $S$ and $Q$, the adaptation constructs the execution traces of an adapter $A$ (i.e., $SET(\overline{A})$), and then the workflow of $A$. The interaction of $A$ with $S$ and $Q$ is then validated through a reachability analysis of the workflow corresponding to their aggregation. Note that, if the composite has at least one lock-free trace, the adaptation process outputs the contracts of $A$, and of the composition between $S$ and $A$.

It is worth noting that our aggregation and adaptation methodology can be successfully employed to generate replaceability adapters, viz., adapters that wrap Web services so that they become compliant with other services (e.g., wrapping new service versions for backwards compatibility). Given two services, $S$ and $S^*$, wrapping $S^*$ so as to behave like $S$ with respect to clients $C$ can be achieved by computing $SET(A)$ as the merge of $SET(S_C)$ and $SET(S^*_C)$. Furthermore, behavioural service customisation, viz., the generation of adapters that wrap services $S^*$ into exposing to clients $C$ a partial behaviour $S$, can be achieved again by computing $SET(A)$ as the merge of $SET(S_C)$ and $SET(S^*_C)$.
Chapter 5

Translating BPEL processes into YAWL workflows

The availability of different languages for the description of Web service behaviour hinders automated Web service location, aggregation, and adaptation, as currently there are no available tools for the automated translation of service protocols.

In this Chapter we present the specification of a translator of BPEL processes into YAWL workflows (BPEL2YAWL, for short).\(^1\) We chose BPEL since it is currently the most widely adopted approach for expressing the behaviour of Web services. BPEL describes business processes through the specification of control and data logic around a set of (WSDL) Web service interactions.

One should note that the BPEL2YAWL translator easily plugs into the aggregation and adaptation methodology and gives clients (e.g., service developers) the possibility to automatically generate the YAWL behaviour information level of the service contracts corresponding to the input business processes. Hence, we argue that the BPEL2YAWL translator contributes towards the semi-automated generation of service contracts from real-world service descriptions. Moreover, as we shall see in the following, the pattern-based compositional nature of the translator simplifies the development of an inverse YAWL2BPEL translator, needed for the deployment of service contracts as Web services.

Roughly, a BPEL process is constructed by wrapping basic activities into structured ones. The basic activities are used, for example, to exchange messages among the services involved in the business process, to delay the execution of the process, or to signal faults. The control-flow in BPEL is achieved, on the one hand, through structured activities such as sequences and switches, and on the other hand, through the use of links to synchronise activities executed in parallel. It is important to note that the semantics of activity execution in BPEL is not straightforward, mainly due to the synchronisation links and to the use of scopes, which wrap activities and provide them with event, fault, and compensation handlers. Our work is also motivated

\(^1\)Preliminary versions of the translator can be found in [18, 24].
by the fact that most approaches that attempt to provide a formal (e.g., Petri net) semantics to BPEL processes [5, 41, 43, 51] do not tackle BPEL synchronisation links and/or the exceptional behaviour of a business process and/or do not take into account data-flow aspects (e.g., transition guards of synchronisation links, or join conditions of activities).

We present a compositional translation based on YAWL patterns. Basically, we define a YAWL pattern for each BPEL activity, as well as for the whole BPEL process. In more detail, we define a Basic Pattern Template (BPT) and a Structured Pattern Template (SPT) to translate basic and structured activities, respectively. The role of patterns is twofold – they provide a unique representation of activities, and they provide an execution context for them.

Given a BPEL process, the BPEL2YAWL translator automatically generates its YAWL translation by:

- Instantiating the pattern of each activity defined in the BPEL process, and by
- Suitably interconnecting the obtained patterns into the final workflow.

Patterns are linked using three types of lines: green lines – to represent the structural dependencies among activities, blue lines – to translate the synchronisation dependencies, as well as red lines – for the propagation of faults toward fault handlers.

To the best of our knowledge, our translator is the first attempt to translate BPEL processes into YAWL workflows. Its main features can be summarised as follows:

- It provides an automated pattern-based compositional translation of BPEL processes into YAWL workflows,
- It copes with all types of BPEL activities (including flows with synchronisation links, and scopes),
- It handles exceptional behaviour – events, faults and (explicit) compensation,
- It straightforwardly plugs into our Web service aggregation and adaptation methodology (described in Chapters 3 and 4),
- The pattern-based compositional nature of the BPEL2YAWL translator sets the basis for the development of an inverse YAWL2BPEL translator, and
- It sets the basis for the formal analysis of BPEL processes.

We also argue that the specification of the translator complements [13] by providing a lightweight semantics of BPEL processes in terms of YAWL workflows. As we shall illustrate in Section 5.2, almost all BPEL activities are provided with simple
intuitive translations in terms of (YAWL) workflows. Hence, the description of the translation also provides an intuitive description of BPEL features.

The Chapter is organised as follows. Section 5.1 briefly introduces BPEL. Section 5.2 is devoted to the specification of the BPEL2YAWL translator. Subsections 5.2.1 and 5.2.2 define the patterns used for translating the BPEL basic and structured activities, respectively, while Subsection 5.2.3 describes the translation of BPEL processes. In Section 5.3 we thoroughly present a simple translation example. Finally, Section 5.4 briefly reviews related work, while some concluding remarks are drawn in Section 5.5.

5.1 A Brief Introduction to BPEL

BPEL [13] is a language for expressing the behaviour of a business process through the specification of control and data logic around a set of Web service interactions. Basically, a BPEL process orchestrates the operations offered by the partner Web services through WSDL [100] interfaces, and in turn, it exposes a WSDL interface to clients.

A BPEL process can be either abstract, or executable. Abstract processes hide implementation details (i.e., private information), while executable processes describe the full interaction behaviour.

BPEL defines the notion of partner link to model the interaction between a business process and its partners. A partner link refers to at most two WSDL port types, one of the interface to the business process (viz., operations offered by the process to the partner), and the other belonging to the interface of a partner (viz., operations offered by the partner to the business process).

BPEL is a hybrid language that combines features from both the block-structured language XLANG [80] and from the graph-based language WSFL [101]. The former contributed with basic activities (e.g., for sending and receiving messages, for waiting for a period of time) as well as with structured ones (e.g., sequential or parallel execution of activities, activity scoping) for combining activities into complex ones. The latter brought the definition of links to synchronise activities executed in parallel. Roughly, the execution of an activity that is the target of synchronisation links is delayed until all activities from where the links emerge are executed. Other features of BPEL are the instance management through correlation sets, event and fault handling, as well as compensation capabilities. The correlation sets are used to identify the various sessions that a business process can have with its clients. Event, fault and compensation handlers make the exceptional behaviour of a business process. Event handlers define message and alarm events, while fault handlers catch and process faults raised in the process. Furthermore, compensation handlers provide roll-back activities to compensate for faults in the process. More details on these topics will be given in Sections 5.2 and 5.3.

The BPEL basic activities are: receive/reply through which a BPEL process
inputs/sends a message from/to a partner service, *invoke* through which a BPEL process asynchronously/synchronously invokes an operation of a partner service, *wait* for delaying the execution of a process, *throw* for signalling faults, *terminate* for explicitly terminating the execution of a process, *empty* for doing a “no-op”, *assign* for copying values between variables, and *compensate* for invoking compensation handlers.

The structured activities are: *sequence*, *switch*, and *while* for sequential, conditional and repeated activity execution, *flow* for parallel activity execution, *pick* for managing the non-deterministic choice of the activity to be executed, and *scope* for providing an execution context for an activity.

For example, consider the following simplified BPEL process (snippet) that computes the greatest common divisor (GCD) of two numbers.

```xml
<process name="GCD" suppressJoinFailure="yes">
  <faultHandler>
    <catch fault="negNum">
      <reply fault="negNum"/>
    </catch>
  </faultHandler>
  <flow>
    <receive(a,b) createInstance="yes">
      <source link="RCV2THR" transitionCondition="a<=0 or b<=0"/>
      <source link="RCV2WHL" transitionCondition="a>0 and b>0"/>
    </receive>
    <throw fault="negNum">
      <target link="RCV2THR"/>
    </throw>
    <while condition="a!=b">
      <source link="WHL2SEQ"/>
      <target link="RCV2WHL"/>
      <scope>
        <faultHandler>
          <catch fault="dec a">
            <assign a:=a-b/>
          </catch>
          <catch fault="dec b">
            <assign b:=b-a/>
          </catch>
        </faultHandler>
        <switch>
          <case condition="a>b">
            <throw fault="dec a"/>
          </case>
          <otherwise>
            <throw fault="dec b"/>
          </otherwise>
        </switch>
      </scope>
      <sequence>
        <assign c:=a/>
        <reply(c)/>
      </sequence>
    </while>
  </flow>
</process>
```

The GCD process defines a *flow* activity, which consists of four activities: a
receive, a throw, a while, and a sequence. Furthermore, the flow defines three synchronisation links. The first two, RCV2THR and RCV2WHL, emerge at the receive activity and target the throw and the while activities, respectively. The third one, WHL2SEQ emerges at the while and targets the sequence. It is important to note that each BPEL activity that is the target of at least one synchronisation link has a (possibly default) joinCondition logical expression that computes the synchronisation status based on the statuses of the input links. Furthermore, the suppressJoinFailure attribute serves for deciding the control-flow in case of a false joinCondition. If the suppressJoinFailure is set to yes, the BPEL engine simply skips the respective activity in order to achieve the dead-path-elimination (see Subsection 5.2.1). Otherwise, the BPEL engine raises a joinFailure fault.

At run-time, the execution of the flow structurally enables its four child activities, yet only the receive can be executed first as the other three activities are constrained from the synchronisation viewpoint (i.e., the statuses of their input links are not known). The receive inputs two numbers, a and b. On the one hand, if both numbers are greater then zero, the BPEL engine sets a negative status for the RCV2THR link and a positive status for the RCV2WHL link. As a consequence, the throw is skipped, and since the suppressJoinFailure attribute is set to yes for the entire BPEL process, a joinFailure fault is not signalled.

The activity to be executed next is the while, which checks whether a is equal to b. If this is not the case, the process continues with the execution of the scope activity inside the while. The scope further consists of a switch, with two branches. If a is greater than b a dec\_a fault is raised. Otherwise, the BPEL engine raises a dec\_b fault. Both faults are to be caught by the faultHandler of the scope activity. In the former case, a is decreased by b (in the first assign activity), while in the latter case b is decreased by a. At this point the scope terminates and the execution of the process continues by checking whether a is now equal to b. If so, a new while cycle is performed. Otherwise, the while terminates and BPEL sets a positive status for the WHL2SEQ link. Consequently, BPEL executes the sequence that first stores the value of a into a new variable c and then it sends it to the invoker of the GCD process through the reply activity.

On the other hand, if at least one of the two numbers is negative or zero, RCV2THR gets a positive status, while RCV2WHL a negative one. Consequently, the BPEL engine executes the throw, during which the while activity is skipped. The execution of the throw raises a negNum fault that is caught by the fault handler of the process, which forwards it to the invoker of the business process through the reply activity.

YAWL was briefly introduced in Chapter 2. More details on the two languages will be discussed in the next Section, while describing the specification of the translator.
5.2 From BPEL to YAWL

The objective of this Section is to present the specification of a translator of BPEL processes into YAWL workflows – with a special care to preserve the information in the BPEL processes so as to make possible the definition of an inverse YAWL2BPEL translator. First, we define a YAWL pattern for each BPEL activity, as well as for the entire business process. Then, the workflow corresponding to a BPEL process is obtained by suitably instantiating and interconnecting the workflows of all its activities.

In Subsection 5.2.1 we first introduce the Basic Pattern Template, and then we uses it to define the patterns of the basic activities. Then, in Subsection 5.2.2 we define the Structured Pattern Template, which we use to define the patterns of the structured activities. Finally, in Subsection 5.2.3 we define the Process pattern template, and we describe the process of obtaining the final workflow.

In the following we shall use the term Pattern Template to refer to the pattern of a generic BPEL activity (viz., either basic or structured). The role of a pattern template is twofold:

- It provides the necessary elements for uniquely identifying an activity/process, as well as
- It provides an execution context for the translated activity/process.
5.2. FROM BPEL TO YAWL

5.2.1 Patterns of BPEL Basic Activities

BPEL uses structured activities to specify the order in which activities have to be executed. For example, the second activity in a sequence can be executed only when the first one has finished its execution. Moreover, the flow construct allows for synchronisation links to be defined among activities. As previously mentioned, when an activity is structurally enabled, BPEL waits for the statuses of all its incoming links (if any) to be determined. At that point BPEL computes the joinCondition (a logical expression), which guards the execution of the activity. A true value leads to the execution of the activity, while a false value leads to either raising a joinFailure fault, or to skipping the entire activity. It is important to note that a structured activity that is skipped leads to skipping all the activities nested within it. Skipping an activity leads to propagating negative (viz., false) statuses on its output links. This process is called dead-path-elimination.

We model the structural relations among BPEL activities through what we call green lines. A pattern has one or more green inputs, which are used to enable it from the structural point of view. Dually, it has one or more green outputs, to be sent upon completion of the pattern, which will be used to enable further patterns. For example, the patterns translating child activities of a BPEL sequence have to be linked through green lines. The pattern corresponding to the first activity in the sequence outputs a green line that is taken as input by the pattern of the second activity in the sequence (in lexical order, since this is the order of execution of the activities in the sequence). Then, the process of linking the patterns of the activities in the sequence through green lines is repeated until the last activity in the sequence.

As we shall see in Subsection 5.2.2, the pattern of the first activity in the sequence inputs a green line from a special pattern that marks the beginning of the sequence pattern. Dually, the pattern of the last activity in the sequence outputs a green line to another special pattern that marks the end of the sequence pattern.

On the other hand, we model the synchronisation links among BPEL activities using blue lines. A pattern has one blue input for each synchronisation link that targets the activity it translates. Analogously, it has one blue output for each link that emerges from the activity it translates. For example, inside a BPEL flow, a synchronisation link from activity A to activity B is translated into a blue line from the pattern translating A to the pattern translating B. Then, the pattern of A is in charge of computing the status of the respective link in a (global) variable, while the pattern of B first waits to receive a blue token on the respective link, and then it computes the value of the joinCondition.

Finally, in order to cope with faults we use red lines. Patterns that treat errors (viz., faults) have red inputs, while patterns that generate errors have red outputs. For example, the translation of the BPEL throw activity has a red line as output, while the translation of the BPEL fault handler inputs one.

The Basic Pattern Template is illustrated in Figure 5.1. It consists of an Execution Prerequisites Block and of an Execution Logic Block. Green input lines of a pattern
are denoted by $gi$, and green outputs by $go$. Similarly, $bi$ and $bo$ denote blue inputs and outputs, and $ri$ and $ro$ red ones.

The Execution Prerequisites Block (EPB). The EPB is in charge of enabling the pattern. In order to execute, a pattern has to be enabled both from the structural and from the synchronisation point of view.

The GreenGate task of the EPB is in charge of waiting for the green tokens. It also inputs a $parentSkip$ boolean variable from its parent\(^2\) activity, whose value indicates whether the latter has been skipped or not. Indeed, since each structured activity could be skipped if it is the target of a synchronisation link, it outputs a $parentSkip$ variable to all the patterns corresponding to its nested (child) activities. If $parentSkip$ holds true then the pattern must be skipped, as one of its ancestors was skipped. In this case GreenGate will immediately enable the Execution Logic Block, without having to wait for the statuses of its incoming links to be computed. If instead $parentSkip$ holds false, then the pattern is ready to be executed from the structural viewpoint. Consequently, the execution of the EPB continues with the BlueGate task, which waits for all blue tokens and then it computes the value of the $joinCondition$ by taking into account the statuses of its incoming links stored into $bi$ boolean variables.\(^3\) Then, the BlueGate enables the Execution Logic Block. Note also that the Green Gate is in charge of setting the $act.X.completed$ (global) variable to false, indicating that the pattern was not successfully executed yet.

The Execution Logic Block (ELB). The ELB has three possible execution scenarios: It can execute successfully, it can be silently skipped, or it can raise a fault. While the first and the second case correspond to executing and skipping, respectively, the pattern, the third behaviour corresponds to a false $joinCondition$ (see next) or to an erroneous execution of the activity.

The ExecOrSkip task of the ELB computes the skipping condition (into the $skip$ boolean variable) as a logical disjunction between the $parentSkip$ and the negation of the $joinCondition$ variables. Indeed, an activity is skipped either since one of its ancestors was skipped ($parentSkip = true$), or since its $joinCondition$ is false. If $skip$ evaluates to false, the ActivitySpecificTask is executed, otherwise the ComputeTransitionConditions task is executed.

The ActivitySpecificTask is the key task of the pattern. It uniquely identifies the translated activity and it provides the computations needed by the activity. Instantiating the Basic Pattern Template for a particular activity consists of equipping the

---

\(^2\)When an activity $A$ is directly nested within a structured activity $S$, we also say that $S$ is the parent of $A$ and that $A$ is a child of $S$.

\(^3\)Note that BPEL uses (possibly default) transitionConditions for the synchronisation links, as well as (possibly default) joinConditions for each activity that is the target of at least one synchronisation link. As a consequence, the $joinCondition$ defined by a BlueGate task corresponds to the BPEL $joinCondition$ except that the statuses of the synchronisation links are replaced by corresponding $bi$ variables.

\(^4\)Assume that $X$ is an unique identifier of the activity.
ActivitySpecificTask with a name identifying the activity, and with the inputs and outputs defined by the activity. For example, the Wait pattern has an ActivitySpecificTask called Wait that inputs the duration of the delay, or the time threshold, similarly to the BPEL wait.

The execution of the ActivitySpecificTask is simulated through the deferred choice consisting of the Fault and Success tasks, together with their input place. The environment (viz., the client of the workflow) determines whether Fault or Success is executed. The execution of the Fault task corresponds to an erroneous execution of the activity (e.g., a receive activity has received an incorrect message). The Fault task outputs the name and data associated with the fault, and it sets the boolean fault flag to true. Note that the name of the fault can be either a standard fault given in “Appendix A – Standard Faults” of BPEL 1.1, [13], or an user-defined one (in the case of throw activities). Success corresponds to a successful execution of the activity, and it sets to true the act_X_completed (global) variable, indicating that the pattern was successfully executed. (For simplicity, note that we shall not represent this variable in the patterns built on top of the Basic Pattern Template.) It is important to note that the deferred choice must be defined only for activities whose execution may be erroneous (e.g., receive, invoke). Otherwise, the ActivitySpecificTask is to be directly connected to the ComputeTransitionConditions task (e.g., the Empty pattern template, Wait).

BPEL uses the suppressJoinFailure attribute to determine the process' behaviour when the joinCondition is false. If the suppressJoinFailure attribute corresponding to an activity (defined by it or by one of its ancestors) is set to no, the BPEL engine raises a joinFailure fault. Otherwise, the activity is silently skipped (as if an empty were executed) and BPEL employs the dead-path-elimination by propagating negative statuses on all its output links. The ComputeTransitionConditions task concludes the execution of the ELB and of the pattern. On the one hand, it signals a joinFailure by setting the fault flag to true in case of a false joinCondition if the corresponding suppressJoinFailure attribute is set to no (viz., fault = fault or (not joinCondition and (suppressJoinFailure = NO))). Note that a red output link is to be defined for a pattern that does not employ the deferred choice (viz., that does not raise faults implicitly) if and only if the suppressJoinFailure corresponding to the activity being translated is set to no because, otherwise, the red line is redundant.

We recall that the only pattern that inputs red lines is the pattern corresponding to the BPEL fault handler, which serves for catching and processing faults raised in the process. We shall describe the Fault Handler pattern in Subsection 5.2.2. On the other hand, ComputeTransitionConditions computes the status of each output (synchronisation) link, as defined by the transitionCondition attribute of the respective BPEL link. Link statuses are stored into bo variables, which have to be mapped onto bi variables of other patterns when constructing the workflow of the business process (viz., bo_k = (not skip) and transitionCondition(bi_k)).

Upon completion, the ELB outputs green and blue tokens if and only if the pattern was successfully executed. Dually, it outputs a red token if a fault was
raised.

In order to obtain the pattern template of a basic activity, one has to:

1. Customise the ActivitySpecificTask,

2. Remove the deferred choice controlling the success of the activity if the activity cannot have an erroneous execution, and

3. Set the (maximum) number of inputs and outputs of the pattern.

The customisation of the ActivitySpecificTask regards the name of the task, which has to identify the pattern, as well as the inputs and the outputs of the task, which are obtained from the inputs and the outputs of the BPEL activity. Other possible modifications may involve the removal of the BlueGate, the employment of guards for the BlueGate, and so on, as we shall see in the following. Furthermore, note that a pattern always has at least one green input and one green output.

In the following we describe the pattern templates corresponding to the BPEL basic activities. Note however that we translate the assign and the compensate using structured pattern templates, due to the execution semantics of the two activities. Both patterns will be described in Subsection 5.2.2.

Empty

The BPEL empty activity has the following form:

```xml
<empty standard-attributes>
  standard-elements
</empty>
```

where the standard-attributes are:

- `name=“ncname”`?
- `joinCondition=“bool-expr”`?
- `suppressJoinFailure=“yes|no”`?

and the standard-elements are:

```xml
<source linkName=“ncname” transitionCondition=“bool-expr”?/>*  
<target linkName=“ncname”/>*
```

An empty activity performs a “no-op”, and it may be useful e.g., inside fault handlers to suppress caught faults, or as milestones inside flow activities. The `joinCondition` is a boolean expression constructed using the statuses of the incoming synchronisation links as operands, and a false `joinCondition` leads to skipping the activity. If the `suppressJoinFailure` attribute is set to `yes`, then the activity is silently skipped, otherwise a `joinFailure` fault is raised by the BPEL engine. Source tags define synchronisation links emerging from the activity, and the statuses of the
respective links are to be given by \( transitionCondition \) boolean expressions. Dually, an activity can be set as target of a synchronisation link using the \( target \) element. Note that both, the standard attributes and the standard elements, are optional.

The Empty pattern (see Figure 5.2) is the simplest pattern. It does not contain the deferred choice block (consisting of the YAWL condition together with the Fault and Success tasks) as the execution of an empty activity cannot raise an explicit fault. Consequently, the Empty task is directly connected to the ComputeTransitionConditions task. Furthermore, Empty does not employ any inputs and outputs (IOs) because the empty activity does not define any variables. Note that the Empty pattern has one green input and one green output only, because from the structural viewpoint an empty activity has one predecessor and one successor only.\(^5\)

**Receive**

The BPEL receive has the following form:

```xml
<receive partnerLink="ncname" portType="qname" operation="ncname"
    variable="ncname"? createInstance="yes/no"?
    standard-attributes>
    standard-elements
    <correlations>?
        <correlation set="ncname" initiate="yes/no"?>
        </correlation>
    </correlations>
</receive>
```

BPEL uses receive activities to input messages either from the invoker of the BPEL process, or from partner Web services. Roughly speaking, the receive specifies the partnerLink it expects to receive from, and the portType and operation that it expects the partner to invoke. The variable (used to store the received message) and the createInstance (used to instantiate the business process) attributes are both

\(^5\)Although some constructs are redundant, e.g., the AND-join of the GreenGate task, we shall keep them in the patterns in order to simplify the description of the translator.
optional. If the createInstance is set to yes, then the reception of a message by the receive leads to starting a new process instance. By default, the createInstance is set to no. Since a business process typically holds one or more conversations with its partners, BPEL uses correlation sets to route the messages involved in a conversation to the correct service instance. For example, the various conversations a seller process holds with its buyers may be distinguished by using e.g., the purchase order number (supplied by buyers at the initiation of the conversation) as correlation token. We shall not go into further details on the correlation of business process instances, since YAWL does not model multiple workflow instances in this dynamic way. In YAWL, the clients of a workflow are in charge of (manually) starting a new workflow instance (called workflow case) by instantiating a workflow specification in the YAWL engine. As [86] notes, the engine handles the execution of these cases, i.e. based on the state of a case and its specification, the engine determines which events it should offer to the environment. However, the translator we propose in this Chapter imports the correlation sets (and other information that is strictly related to, and mandatory only for the execution of the BPEL process, such as partnerLinks, portTypes, and so on) into the YAWL workflow as (global) variables, which are useful e.g., for an inverse YAWL2BPEL translator that deploys YAWL workflows as BPEL processes.

The pattern of the receive activity is given in Figure 5.3. As expected, the ActivitySpecificTask is now called Receive. It inputs the partnerLink, portType, and operation, as well as the optional variable, createInstance, and correlationSet attributes of the receive activity. Note that instantiating the Receive pattern template for a particular business process to be translated, resumes to “hard-coding” all the inputs of the Receive task but the variable one, with the values of the corresponding attributes in the receive activity. The variable input receives a value at run time,
which logically corresponds to the value inputted by the receive activity in the BPEL process. The Receive has one green input only (coming from the pattern of the activity structurally preceding it), and it can employ up to two outputs: one (mandatory) for the pattern of the activity structurally following it, and another (optional) used to enable the pattern for event handling of the entire business process. (More details on this later, when describing the pattern of the BPEL process.) Please note that the second (optional) output should be used only if the BPEL process to be translated has an event handler at the process level and if the createInstance attribute of the receive activity is set to yes. Consequently, a green token is sent on the second green output provided the following logical condition holds: 

\[
\text{"(skip = F) and (createInstance = YES) and (somebodyCreatedAnInstance = F)"}
\]

We use the somebodyCreatedAnInstance (global) variable in order to avoid multiple green tokens being sent to the pattern for event handling of the entire business process by multiple Receive patterns. For example, assume that the business process consists of a flow having two receive operations. In this case the first receive to be executed enables the event handler of the process.

Note that a Receive can raise an error either due to the execution of the Fault task (e.g., corresponding to a mismatch between the types of the expected message in the receive and of the message sent by the invoker of the business process), or due to skipping the Receive pattern when its corresponding suppressJoinFailure is set to no. In both cases, a red token is generated by the Receive pattern.

Reply

The BPEL reply has the following form:

```xml
<reply partnerLink="ncname" portType="qname" operation="ncname"
variable="ncname"? faultName="qname"? standard-attributes>
standard-elements
<correlations>?
  <correlation set="ncname" initiate="yes|no"?>
  </correlation>
</correlations>
</reply>
```

BPEL uses reply activities to send response messages to requests previously accepted through receive activities. (The combination of a receive and a reply forms a request-response operation on the WSDL portType of the process.) Similarly to a receive, the reply is identified by the triple (partnerLink, portType, operation), and it may use correlation sets to identify the conversations with its business partners. A reply activity can be used to send back to the invoker of the synchronous operation either a normal response message, or a fault message. On the one hand, a normal response does not include the faultName attribute, and the variable attribute (if present) provides the data to be sent to the partner. On the other hand, the fault message is identified through the use of the faultName attribute, and the variable attribute (if present) gives the fault data.
The Reply pattern template is presented in Figure 5.4. The inputs of its Reply task are the partnerLink, portType and operation, as well as the optional correlationSet attributes of the reply activity, while its output is either a variable or a faultName, or both a variable and a faultName, as defined in the reply activity. Note that the Reply pattern does define the deferred choice construct since the execution of a reply activity cannot raise a fault. Note however that a Reply pattern can raise a fault as a consequence of skipping the pattern when its corresponding suppressJoinFailure attribute is set to no. In such case a red token is generated by the Reply pattern. Finally, a Reply has one green input and one green output.

Invoke

The BPEL invoke has the following structure:

```xml
<invoke partnerLink="ncname" portType="qname" operation="ncname"
    inputVariable="ncname"? outputVariable="ncname"?
    standard-attributes>
    standard-elements
    <correlations>?=
        <correlation set="ncname" initiate="yes|no"? pattern="in|out|out-in"/>
    </correlations>
    <catch faultName="qname" faultVariable="ncname">*
        activity
    </catch>
    <catchAll>*
        activity
    </catchAll>
    <compensationHandler>*
        activity
    </compensationHandler>
</invoke>
```

A BPEL process can invoke operations offered by the partner Web services. An asynchronous invocation (viz., of a one-way WSDL operation) requires only the
5.2. FROM BPEL TO YAWL

The synchronous Invoke Pattern Template

![Diagram of the synchronous Invoke Pattern Template]

Figure 5.5: The Invoke pattern templates.

\[ \text{inputVariable} \] to be defined, while for a synchronous invocation (viz., of a request-response WSDL operation) the \text{invoke} should define both \text{inputVariable} and \text{outputVariable}. Similarly to the receive and reply operations, the \text{invoke} may use correlation sets. Note that a synchronous invocation returning with a WSDL fault (see reply before) can be caught locally by the \text{invoke} through the inline catch/catchAll, which will execute the activity it contains. Moreover, the \text{invoke} may define a “rollback” activity through the inline compensation handler. However, the invoke with the inline fault and compensation handlers is semantically equivalent to the same \text{invoke} activity enclosed in a \text{scope} that defines the respective fault and compensation handlers. (See the translation of the \text{scope} structured activity in Subsection 5.2.2 for more information on the fault and compensation handlers.) Hence, in order to simplify the translation we shall treat \text{invoke} activities with inline fault and compensation handlers as \text{scopes} immediately enclosing the \text{invokes} and providing these handlers.

Similarly to the Receive and Reply pattern templates, the asynchronous Invoke (depicted in the top-part of Figure 5.5) has an Invoke task that inputs a partnerLink, portType, and operation, as well as (optional) correlationSet variables. When instantiating it for the translation of a particular BPEL process, a correlationSet variable is defined for each correlation attribute of the \text{invoke} activity in the business process. The pattern of an asynchronous Invoke task can specify at most the inVar output variable, and it should be defined by an instance of the Invoke pattern template (i.e., an Invoke translating an asynchronous invoke activity in a particular BPEL process) only when the corresponding attribute exists in the \text{invoke} activity
being translated.

The bottom-part of Figure 5.5 describes the pattern of a synchronous invoke activity, which is similar to the pattern of a BPEL sequence (see Subsection 5.2.2). Roughly, the synchronous Invoke pattern consists of two patterns linked in a sequence. On the one hand, Begin(Invoke) is similar to the pattern of the asynchronous invoke activity, however it does not output blue tokens, since the invocation only terminates after the execution of the End(Invoke) pattern. On the other hand, End(Invoke) basically differs from the pattern of the asynchronous invoke activity in that it does not input blue tokens. Furthermore, its End(Invoke) task may input an outVar variable. Similarly to a Receive, End(Invoke) can output red tokens e.g., in case of a mismatch between the expected message and the actual received one. For more information on the semantics of the structured pattern templates please see Subsection 5.2.2.

Wait

The BPEL wait:

```xml
<wait (for="duration-exp" | until="deadline-exp") standard-attributes>
standard-elements
</wait>
```

delays the execution of a business process either for a certain period of time (through the for attribute) or until a certain deadline is reached (through the until attribute).

The Wait pattern template is given in Figure 5.6. Its construction is identical to the Empty pattern with the exception of the ActivitySpecificTask. The Wait pattern template allows the Wait task to have either a for or a until variable, depending on the corresponding attribute defined in the business process to be translated. (The only particularity of the Wait task with respect to all other ActivitySpecificTasks
is that it invokes the YAWL *TimeService* in order to delay the execution of the workflow.)

**Throw**

The structure of the BPEL *throw* is the following:

```
<throw faultName="qname" faultVariable="ncname"? standard-attributes>
  standard-elements
</throw>
```

A *throw* activity serves for explicit fault signalling. Each fault is defined by a (unique) *faultName* and an (optional) *faultVariable* containing the fault data.

The **Throw** pattern template is illustrated in Figure 5.7. It employs a *Throw* task that outputs a fName variable and possibly a fVar variable corresponding to the *faultName* and, respectively, *faultVariable* attributes in the BPEL process to be translated. It is important to note that **Throw** outputs a red token, either if a *joinFailure* fault is being raised (viz., *fault* = T), or if the **Throw** task is executed (viz., *skip* = F). In both cases the **Throw** pattern only outputs one red token. When the **Throw** is successfully skipped due to a *suppressJoinFailure* attribute set to *yes* (viz., a *joinFailure* is not raised in this case), the pattern outputs the green skipping token, and the blue tokens.

**Terminate**

**Terminate** activities are defined as follows:

```
<terminate standard-attributes>
  standard-elements
</terminate>
```
A terminate activity is used to end the execution of the entire business process instance. All running activities are to be terminated immediately without any fault or compensation handling. The semantics of activity termination [13] depends on the activity to be interrupted. For example, assign activities are allowed to finish their execution, while wait activities are ended immediately. Although it is not trivial, one can obtain this behaviour in the translated YAWL workflow by suitably equipping the pattern corresponding to the end of the business process (see End(Process) in Subsection 5.2.3) with a cancellation set including only the activities whose execution has to be interrupted. However, in order to keep the translation simple, our translator adds the entire Pattern Template corresponding to the activity defined by the process into the cancellation set of End(Process).

The pattern template of the BPEL terminate (see Figure 5.8) is quite similar to the Empty pattern. However, Terminate outputs only one green token on one of its two green outputs. If the Terminate is skipped without raising a joinFailure (i.e., “fault = F and skip = T”), then a green token is sent to the pattern translating the activity structurally following the terminate in the BPEL process. Otherwise, if the Terminate is executed (i.e., “fault = F and skip = F”) then the green token is sent to End(Process) in order to cancel the execution of the entire business process.

For simplicity, the BPEL assign and compensate activities are translated into structured patterns, as we shall see in the next Subsection. On the one hand, the BPEL assign may contain several copy tags each signifying a data exchange that may lead to a fault being raised. Hence, we treat the assign similarly to a sequence activity. On the other hand, the BPEL compensate leads to the execution of the activity defined in the compensation handler of the scope to be invoked. As a result, the compensate finishes its execution when the activity in the invoked compensation
handler has finished its execution. This leads to the need of explicitly representing
the beginning and the end of the compensate, and consequently we treat it as a
structured activity.

5.2.2 Patterns of BPEL Structured Activities

A BPEL structured activity defines one or more activities to be executed in a certain
order. In order to cope with this, we define the Structured Pattern Template as a
tuple consisting of a Begin pattern, an End pattern, as well as a Pattern Template
for each child activity.

The purpose of the Begin and End patterns is to provide an identification for
the activity being translated. More importantly, the execution of Begin logically
corresponds to the initiation of the structured activity (as a whole), whereas the
execution of End logically marks the termination of the structured activity. Con-
sequently, note that the Begin pattern is in charge of setting the act_X_completed
variable to false, while the successful execution of the End pattern (viz., the execu-
tion of the End task – see next) has to set the act_X_completed variable to true,
indicating that the structured pattern was successfully executed. Both Begin and
End patterns are generated from the Basic Pattern Template, and they are quite
similar to the Empty pattern. On the one hand, Begin is in charge of enabling the
structured pattern both from the structural and synchronisation viewpoints. Hence,
Begin has to input the green and the blue lines and to raise a joinFailure in case of
a false joinCondition if the corresponding suppressJoinFailure attribute is set to no.
Furthermore, it provides a green output for each Pattern Template corresponding
to a child activity that can be executed first. On the other hand, End has to wait
for the green tokens from all Pattern Templates of the child activities that have to
be executed last. Moreover, End is the source of the blue outputs corresponding
to synchronisation links having as source the structured activity. In general, End
cannot lead to any faults being raised, and hence it does not have a red output.

A structured activity introduces a new nesting level and consequently Begin has
to output a parentSkip variable to the patterns of all the (child) activities nested
inside the structured one, as well as to the End pattern. In this way we achieve the
dead-path-elimination inside structured patterns.

Now, the pattern templates of all structured activities are obtained by adjusting
the Begin and End patterns and by suitably interconnecting them with the Pattern
Templates. Basically, both processes depend on the way in which the structured
activity enables for execution its child activities. In the following we shall write
Begin(X) and End(X) to refer to the Begin and End patterns of a structured activity
X.

The BPEL structured activities are:

- sequence, switch, and while, that provide sequential control between activities,
- flow, which provides concurrency and synchronisation between activities,
• *pick*, which provides non-deterministic choice based on external events, and
• *scope*, which provides a behaviour context for activities.

Furthermore, as we already mentioned, in this Subsection we shall describe the implementation of the BPEL basic activities *assign* and *compensate*.

The **Sequence**, **Switch**, **Flow** and **Pick** patterns all share the same structure: non-deterministic

\[
\begin{align*}
\text{Sequence} & \rightarrow \text{Begin(Sequence)} \quad \text{PatternTemplate}^+ \quad \text{End(Sequence)} \\
\text{Switch} & \rightarrow \text{Begin(Switch)} \quad \text{PatternTemplate}^+ \quad \text{End(Switch)} \\
\text{Flow} & \rightarrow \text{Begin(Flow)} \quad \text{PatternTemplate}^+ \quad \text{End(Flow)} \\
\text{Pick} & \rightarrow \text{Begin(Pick)} \quad \text{PatternTemplate}^+ \quad \text{End(Pick)}
\end{align*}
\]

The BPEL *sequence* defines one or more activities to be performed sequentially, in lexical order, and it has the following structure:

\[
<\text{sequence standard-attributes}> \\
\quad \text{standard-elements} \\
\quad \text{activity}+ \\
<?,\text{sequence}>
\]

The **Sequence** pattern (see Figure 5.9) is the simplest structured pattern template. **Begin(Sequence)** differs from the **Empty** pattern template in that it does not have blue output links. This is because the statuses of the BPEL synchronisation links emerging from a *sequence* are to be computed upon completion of the *sequence* activity. Consequently, the output blue links translating the emerging synchronisation links (if any) are defined by the **End(Sequence)** pattern template.\(^6\) As **End(Sequence)** logically marks the termination of a *sequence*, it cannot be the target of a synchronisation link, and hence it does not have any blue inputs. Note however the red output of **Begin(Sequence)**, which serves for signalling a *joinFailure* since the synchronisation links targeting a *sequence* activity are translated into blue inputs of the **Begin(Sequence)** pattern template. Furthermore, **End(Sequence)** does not have a red output as its execution cannot lead to faults being raised. On the one hand, the **ExecOrSkip** task of **Begin(Sequence)** computes the value of the *skip* variable as a disjunction between the *parentSkip* and the negation of the *joinCondition*. On the other hand, **End(Sequence)** directly sets the value of the *skip* variable to the value of the *parentSkip*.

The top-part of Figure 5.9 presents the structural dependencies (i.e., how the **Pattern Templates** of the **Sequence** are connected through green lines) among the **Begin(Sequence)** pattern, the **Pattern Templates** translating the child activities of the *sequence*, and the **End(Sequence)** pattern. In the following we shall use the

\(^6\)Note that the **ComputeTransitionConditions** of **Begin(Sequence)** computes only the *fault* flag.
“cloud” symbol as a simplified denotation of a Pattern Template. (We recall that a Pattern Template is used to denote the pattern of a generic BPEL activity.) Note that the cloud representing each Pattern Template is dashed as it may correspond to the translation of a structured BPEL activity (e.g., another sequence), and hence it may contain several other Pattern Templates (i.e., clouds). Begin(Sequence) has one green output only because only one activity can be executed first in a sequence. Consequently, the green output of Begin(Sequence) is linked as input of the Pattern Template translating the first activity in the BPEL sequence. Dually, End(Sequence) employs one green input only, which comes from the pattern of the last activity in the sequence. Furthermore, each Pattern Template translating a BPEL activity in the sequence (except the first and the last ones) has a green input from the Pattern Template of the previous activity in the sequence, and a green output for the Pattern Template of the next activity in the sequence.

Switch

The switch activity consists of one or more conditional branches guarded by boolean expressions as well as an (optional) otherwise branch. The activity to be executed by the switch is determined by the first guard that holds true in lexical order. The
Figure 5.10: The \textbf{Switch} pattern template.

activity corresponding to the \textit{otherwise} branch is executed provided no guard holds. When the \textit{otherwise} branch is not specified, BPEL considers a default \textit{otherwise} enclosing an \textit{empty} activity.

\begin{verbatim}
<switch standard-attributes>
  standard-elements
    <case condition="bool-expr">+activity
      </case>
    <otherwise>?activity
  </otherwise>
</switch>
\end{verbatim}

The pattern template of a BPEL \textit{switch} (illustrated in Figure 5.10) is constructed similarly to the pattern of a \textit{sequence} activity. It is composed of \texttt{Begin(Switch)}, \texttt{End(Switch)}, as well as one or more \texttt{Pattern Templates} for each (child) activity defined by a conditional branch (viz., case) of the BPEL \textit{switch}. \texttt{Begin(Switch)} and \texttt{End(Switch)} are similar to \texttt{Begin(Sequence)} and \texttt{End(Sequence)}, respectively. The former logically marks the beginning of the \textit{switch} and it is in charge of activating the \textit{Switch} pattern by waiting for the green token, as well as for the blue ones (if any). The latter marks the end of the BPEL \textit{switch} and it sets the statuses of its output links (if any).

As previously mentioned, the guards of the \textit{switch} branches are evaluated in the order in which they appear. This is the reason why the \texttt{Switch} pattern template is
constructed by sequentially linking (through the green line) the Pattern Templates corresponding to all conditional branches. The particularity of the Pattern Template that translates an activity defined by a case or otherwise conditional branch is that its GreenGate task has to check whether a previous branch pattern was executed.\textsuperscript{7} Furthermore, the Case patterns have to check further whether the guard condition holds (see the bottom-part of Figure 5.10). As a result, at run-time, if a branch pattern was already executed, or if the guard does not hold, the pattern of the respective branch is skipped in order to employ the dead-path-elimination. As the BPEL specification notes [13], if there is no otherwise branch defined, a default one with an empty activity has to be considered. Consequently, the translator automatically considers for the translation of such switch activities an Empty pattern.

Flow

The BPEL flow provides concurrency and synchronisation inside the business process. Its structure is as follows:

\begin{verbatim}
<flow standard-attributes>
 standard-elements
 <links>?
   <link name=“ncname”>+ 
 </links>
 activity+
</flow>
\end{verbatim}

All child activities of the flow are executed as soon as the flow starts, provided they are not targeted by any synchronisation link. One may note below that the grammar of the flow activity allows for links to be defined. The execution of an activity that is the target of at least one synchronisation link is delayed until the statuses of all of its incoming links are known, and it will be executed only if its corresponding joinCondition holds true. Otherwise, depending on the (corresponding) value of the suppressJoinFailure attribute, the activity is either silently skipped, or a joinFailure is raised. Note that the dead-path-elimination process will forward negative (viz., false) statuses on all the output links (if any) of the activity being silently skipped.

The Flow pattern template (see Figure 5.11) employs similar constructs to the Sequence one. The main difference between Begin(Flow) and Begin(Sequence) is that the former has multiple green outputs, one for each Pattern Template translating a child activity of the flow. Dually, End(Flow) differs from End(Sequence) in that it has multiple green inputs, each coming from a Pattern Template. This is motivated

\textsuperscript{7}Note that the Switch pattern makes use of a caseExecuted variable initially set to no by Begin(Switch), and further set to yes by the branch pattern executed first. In order to avoid the execution of multiple branches, each branch pattern guard simply checks the status of the caseExecuted variable (see the bottom-part of Figure 5.10).
CHAPTER 5. TRANSLATING BPEL PROCESSES INTO YAWL WORKFLOWS

The Flow Pattern Template

Figure 5.11: The Flow pattern template.

by the fact that the execution of a BPEL flow starts by enabling from the structural viewpoint all its children activities. (Consequently, tokens are sent on all its green outputs provided the fault flag is false.) Dually, the flow terminates only when all its child activities have finished their execution. This is achieved through the AND-join of the GreenGate task of End(Flow).

Pick

The grammar of the BPEL pick is given hereafter.

```
<pick createInstance="yes|no"? standard-attributes>
  standard-elements
  <onMessage partnerLink="ncname" portType="qname"
    operation="ncname" variable="ncname"?>+<correlations>?
    <correlation set="ncname" initiate="yes|no"?>+<</correlations>
    activity
  </onMessage>
  <onAlarm (for="duration-expr" | until="deadline-expr")>*
    activity
  </onAlarm>
</pick>
```

A pick defines one or more onMessage elements, as well as optional onAlarm elements. Through an onMessage element the business process waits for a message event from its partner Web services, similarly to a receive. A message event can use correlation sets, as well as it may start a business process instance. Note that, differently from the deterministic choice made by the switch activity and concerning the activity to be executed, the pick makes a non-deterministic choice inside the business process, as the environment decides the activity to be executed next. Furthermore, an onAlarm branch waits for an alarm event to take place, similarly to a wait.
5.2. FROM BPEL TO YAWL

The Pick Pattern Template

**PatternTemplate**

* Each PatternTemplate has to check whether it corresponds to the selected branch.

** Each PatternTemplate of an onMessage branch has two green outputs, and go2 has to be taken as input by the GreenGate of the Begin(EventHandler) of the Process pattern template (if any).

Figure 5.12: The Pick pattern template.

Roughly, the execution of the *pick* resumes to waiting the occurrence of either a message or an alarm event, which leads to executing the activity associated with the event that took place. The occurrence of a message event immediately inactivates the other message events, as well as all the alarms, so that they cannot be triggered. Dually, if an alarm event goes off, all the message events are inactivated, as well as all the other alarms are set off. The *pick* finishes when the activity corresponding to the branch that was triggered terminates.

The high-level view of the Pick pattern template (Figure 5.12) is similar to the one of the flow activity. Begin(Pick), like Begin(Flow), outputs multiple green lines, one for each of the *pick*’s branch activities. However, its BeginPick task is a composite task in charge of branch selection (see the bottom part of Figure 5.12). BeginPick employs one Dummy onMsg_i task for each onMessage branch, as well as one Dummy onAlarm_j task for each onAlarm branch of the *pick*. The execution of the Init task of BeginPick places a token in the condition of the deferred choice as well as it enables the alarm tasks. Note that all Dummy onAlarm_j tasks use (similarly to the Wait task of the Wait pattern) the YAWL TimeService to implement the timer. Although all Dummy onMessage_i are executable due to the token in the deferred choice condition, the first one to be executed (i.e., the first message that arrives)
clears the token. As a consequence, *Dummy onMessage*$_i$ sets the value of the *branch* variable to its identification (which corresponds to the pattern of the activity being triggered). Furthermore, its execution leads to cancelling all timers (see the solid-line cancellation set in Figure 5.12). Then, the *Dummy onMessage*$_i$ task forwards the token to the *Wait4BranchDecision* task, which marks the termination of the *BeginPick* composite task. The other possible execution scenario consists in the completion of a *Dummy onAlarm*$_j$ task when the respective timer sets off before any *Dummy onMessage*$_i$ is executed. The result is that *Dummy onAlarm*$_j$ sets the value of the *branch* variable to its identification, and then it cancels all the other timers and it clears the token in the deferred choice condition (so that no *Dummy onMessage*$_i$ tasks can be executed) – see the dashed-line cancellation sets in Figure 5.12. Finally, the green token reaches the *Wait4BranchDecision* task and the execution of *BeginPick* terminates.

It is important to note that even though only one branch (i.e., the triggered one) will be executed, the *Begin(Pick)* pattern outputs green tokens for all branch patterns in order to achieve dead-path-elimination on the branches that were not selected. The *End(Pick)* pattern template is the same as *End(Flow)*. It just waits for the green tokens from all branch patterns.
Another important characteristic of the **Pick** pattern is the slight modification it brings to the **Pattern Templates** translating its branch activities (see Figure 5.13). Each such **Pattern Template** has to check whether the respective branch was triggered, by comparing its identification (viz., the *IamBranch* variable in the Figure) with the one outputted by the *BeginPick* composite task of *Begin(Pick)* (viz., the *branch* variable in the Figure). The difference between the patterns of message and alarm branch activities is that the pattern for a message branch has to output (similarly to the *Receive* pattern template) a green token to enable the pattern for event handling of the entire business process if and only if the *createInstance* is set to *yes*, if no faults were raised by the branch pattern, if the *Pick* was not skipped and it can create a process instance, and if no other *Receive* or *Pick* branch has already created a process instance. (See also the **Scope** pattern template next.) Furthermore, note that this green output should be defined if and only if the *createInstance* attribute is set to *yes* in the *pick* activity being translated.

One last thing to note about the BPEL *pick* is that if a branch activity *A* of the *pick* is a basic one, then its pattern template is constructed as shown in Figure 5.13, depending on the branch type (message or alarm). Otherwise, if *A* is a structured activity, then its *Begin(A)* and *End(A)* patterns will incorporate the modifications shown in Figure 5.13. (Note that this applies to all other structured pattern templates that bring modifications to the patterns of their children activities.)

### While

The BPEL *while* repeatedly executes its child activity for as long as the boolean while guard holds true. Its structure is the following:

```xml
<while condition="bool-expr" standard-attributes>
  standard-elements
  activity
</while>
```

The **While** pattern

\[
\text{While} \rightarrow \text{Begin(While) PatternTemplate End(While)}
\]

(see Figure 5.14) consists of *Begin(While)*, a **Pattern Template**, as well as *End(While)*, linked in a sequence. The main particularity of the pattern is that *Begin(While)* takes two green inputs – one from the pattern of the activity (structurally) preceding the *while*, and another from *End(While)*. The former is inputted by the *GreenGate* task, while the latter directly enables the **Execution Logic Block** of the **While** as it corresponds to a new iteration and *Begin(While)* should not wait for more tokens on the blue inputs (if any). Note that at run-time only one of the two input green tokens is needed to structurally enable *Begin(While)*. Furthermore, the *ExecOrSkip* task of *Begin(While)* is in charge of checking the loop guard. If the guard evaluates
to true, the While is executed, otherwise it is skipped in order to achieve the dead-path-elimination.\footnote{Note that one cannot check the guard in the GreenGate as the guard may employ variables set by activities that target the while, and hence the verification of the guard has to be done after receiving all blue tokens.}

The particularity of \texttt{End(While)} (with respect to \texttt{End(Sequence)}) is that it has two green output lines, one which goes to the pattern of the activity to be executed next, and another returning to \texttt{Begin(While)}. The \texttt{ComputeTransitionConditions}, in addition to computing the statuses of the blue (synchronisation) output links, checks the loop guard as well. If the guard holds true, \texttt{End(While)} outputs only one green token, on the link to \texttt{Begin(While)}. Otherwise, the green token is sent to the pattern to be executed next. We double-check the guard in \texttt{End(While)} because, if we would simply forward the green token to \texttt{Begin(While)}, a false guard would lead to employing the dead-path-elimination inside the While, which would be incorrect as the loop has already been executed.

Finally, it is important to note that blue outputs are sent by \texttt{End(While)} only when the While terminates, that is, after skipping it (viz., \texttt{skip = true}) or if the loop guard does not hold (viz., \texttt{whileCond = F}).
Assign

The BPEL assign can be used to copy data between variables, to perform simple computations by mapping expressions onto variables, as well as to copy endpoint references to and from partner links. From the assign grammar given below, one may note that an assign may define several copy elements, each one performing an assignment.

```
<assign standard-attributes>
  standard-elements
  <copy>+
    from-spec
    to-spec
  </copy>
</assign>
```

where the from-spec and to-spec have the following structures:

```
<from variable="ncname" part="ncname"/>
<from partnerLink="ncname" endpointReference="myRole|partnerRole"/>
<from variable="ncname" property="qname"/>
<from expression="general-expr"/>
<from> ... literal value ... </from>
```

and

```
<to variable="ncname" part="ncname"/>
<to partnerLink="ncname"/>
<to variable="ncname" property="qname"/>
```

The Assign pattern:

```
Assign → Begin(Assign) Copy+ End(Assign)
```

has the same structure as the Sequence pattern, but it includes Copy patterns rather than arbitrary Pattern Templates (see Figure 5.15). (We recall that we translate the BPEL assign to a structured pattern template due to the fact that an assign may contain several copy attributes, each requiring a data exchange which may lead to a fault being raised.) Begin(Assign) and End(Assign) are identical to Begin(Sequence) and End(Sequence), respectively. Furthermore, due to the fact that BPEL evaluates the assignments in the order in which the copy attributes appear in the assign, we link the Copy patterns (through the structural green line) in a sequence.

The Copy pattern template does not have blue IOs as the BPEL copy can be neither the source, nor the target of a synchronisation link. The assignment is carried out by the Copy task, which maps a (complex) input variable corresponding to the from element in the BPEL copy onto a (complex) output variable corresponding to the to element in the BPEL copy. Hence, an instance of the Copy pattern translating a particular BPEL copy defines the from and to variables of the Copy task depending on the similar attributes of the BPEL copy element (e.g., variable, part, expression).
9 Finally, the Fault task of the Copy pattern can be used to simulate an assignment mismatch, and in such case a red token is outputted by the faulty Copy pattern.

Scope

The BPEL scope is the most complex structured activity, and it employs the following structure.

```xml
<scope variableAccessSerializable="yes/no" standard-attributes>
    standard-elements
    <variables>?
        ...
    </variables>
    <correlationSets>?
        ...
    </correlationSets>
    <faultHandlers>?
        ...
    </faultHandlers>
    <compensationHandler>?
        ...
    </compensationHandler>
    <eventHandlers>?
        ...
    </eventHandlers>
    activity
</scope>
```

9Note that both BPEL and YAWL support XML Schema type definitions and use XPath for data manipulation, and hence translating the BPEL copy into the mapping done by the Copy task is straightforward.
where the handlers are defined as follows:

```xml
<faultHandlers/>
  <catch faultName="qname"? faultVariable="ncname"?>*
    activity
  </catch>
<catchAll/>
  activity
</catchAll>
<compensationHandler/>
  activity
</compensationHandler>
<eventHandlers/>
  <onMessage partnerLink="ncname" portType="qname"
    operation="ncname"
    variable="ncname"*>*\n    <correlations/>
    <correlation set="ncname" initiate="yes|no">+</n    </correlations>
  activity
</onMessage>
<onAlarm for="duration-expr"? until="deadline-expr">*
  activity
</onAlarm>
</eventHandlers>
```

Roughly, a BPEL scope provides a specific context for an activity. It allows for the definition of variables (that live only within the scope) and correlation sets. Furthermore, it contains a (possibly default) optional fault handler, a (possibly default) optional compensation handler, as well as an optional event handler.

The fault handler consists of one or more catch clauses for grabbing faults raised inside the scope. A catch is a container of an activity, guarded by a faultName and an optional faultVariable. The fault handler may also specify a catchAll, which is similar to a catch, yet it does not employ a guard so as to process all faults that reach it. It is important to note that the catches are evaluated in lexical order, and the following rules apply:

1. If the fault has no fault data, BPEL selects the first catch with matching faultName, or the catchAll (if defined). Otherwise,

2. If the fault contains fault data, BPEL selects the first catch with matching faultName and faultVariable. If no such match exists, BPEL selects the first catch with matching faultVariable and no specified faultName, or the catchAll (if defined).

3. If the fault does not match any catch and if there is no catchAll defined, the fault is re-thrown to the immediately enclosing scope. Note further that faults uncaught at the process scope lead to an abnormal process termination, as for a terminate. Moreover, a scope in which a fault occurs is considered to have ended abnormally, even if the fault is processed by the scope’s fault handler.
The compensation handler provides a (compensating) activity that can be invoked either explicitly (through a compensate that specifies the scope to be compensated), or implicitly (during the default compensation mechanism). The compensation handler is activated only when the scope finishes its execution successfully, and consequently, invoking a compensation handler that was not installed is equivalent to a no-op. Note that our translator deals with explicit compensation only, due to the troublesome default compensation mechanism (e.g., the process of compensating a scope inside a while has to invoke the instances of the compensation handler in each successive iteration in reverse order).

Last but not least, an event handler defines message events that can be triggered repeatedly and concurrently during the lifetime of the scope, as well as alarm events that can be triggered at most once while the corresponding scope is active. The former is different with respect to pick activities, which allow only one message event to take place. Note however that when the activity of a scope finishes its execution, the activities running inside an event handler are allowed to complete, yet no other message or alarm events may be triggered. Furthermore, the messages received by the event handler cannot start a business process instance (since the event handler cannot be enabled until the instance is created), as well as the use of the compensate activity inside event handlers is prohibited.

The Scope pattern template has the structure:

\[
\text{Scope} \rightarrow \text{Begin(Scope) PatternTemplate FaultHandler} \ [\text{CompensationHandler}] \ [\text{EventHandler}] \ \text{End(Scope)}
\]

and the structural dependencies among the various patterns involved are illustrated in Figure 5.16. \text{Begin(Scope)} (constructed similarly to \text{Begin(Flow)}) sends green tokens to the Pattern Template translating the (child) activity of the scope, to the
Figure 5.17: The Pattern Template (or Begin(activity)) translating the scope’s activity.

EventHandler, and to the FaultHandler. Furthermore, the BeginScope task of Begin(Scope) sets the FAULT scope variable to no and the scopeActEnded to false. The former is used by the Scope pattern (and in particular by the FaultHandler) to deal with faults being raised inside the BPEL scope, while the latter serves for knowing when the activity defined by the scope terminates so that the EventHandler knows when to finish its execution.

The Pattern Template implementing the scope’s activity (see Figure 5.17) has to set the scopeActEnded variable to true if the scope is executed (viz., not skipped), and it forwards at least one green token. A green (skipping) token is always sent to the GreenGate0 task of the Begin(FaultHandler), while further green tokens are sent to the Pattern Templates belonging to onAlarm patterns (see the EventHandler below), if and only if the scope’s activity was successfully executed (viz., scopeActEnded is set to true). The first one is used to achieve the dead-path-elimination inside the FaultHandler, while the other tokens are used to unlock the onAlarm patterns of the EventHandler after cancelling the respective timers.

After receiving a green token from Begin(Scope), the FaultHandler pattern template further receives either one green token from the Pattern Template (of the scope’s activity) and one green token from the EventHandler (if any), or one red token from the Pattern Template or from the EventHandler. In the former case, the entire FaultHandler will be skipped either because the Pattern Template was completed successfully, or because the entire Scope has to be skipped. The latter case corresponds to a fault being raised (and uncaught) inside the Pattern Template, or inside the EventHandler. If the fault cannot be processed, the FaultHandler sends a green token to End(Scope), which has to output a red token further to the FaultHandler of the parent Scope pattern (if any), or to the FaultHandler of the Process pattern template. Note that only the FaultHandler forwards a (green) token to End(Scope). Furthermore, the FaultHandler pattern template outputs the FAULT variable (as we shall
see later), while the EventHandler inputs the `scopeActEnded` and outputs a `true` value for the `somebodyCreatedAnInstance` variable. When the FaultHandler catches a fault it clears the tokens of the Pattern Template corresponding to the child activity of the scope, as well as the tokens of the activities defined by the event handler.\(^\text{10}\)

End(Scope) (built similarly to End(Sequence)) is in charge of enabling the CompensationHandler when the Pattern Template translating the scope’s activity is executed successfully. (Note that End(Scope) has to save a copy of all the scope variables as required by the CompensationHandler [13].) If the Scope is skipped, End(Scope) has to clear the green tokens received by the FaultHandler from the Pattern Template and from the EventHandler as they are redundant due to the fact that the skipping green token sent by Begin(Scope) to the FaultHandler pattern reaches it first. Dually, the cancellation set of End(Scope) should clear red tokens stuck at the Begin(FaultHandler) pattern in case multiple faults reach it before it is executed. Furthermore, in this case it is unnecessary to perform the dead-path-elimination inside the EventHandler as links cannot cross its boundary. However, we do have to perform the dead-path-elimination inside the FaultHandler.

FaultHandler

The FaultHandler pattern has a similar structure to the Sequence pattern (see Figure 5.18):

\[
\text{Begin(FaultHandler)} \quad \text{PatternTemplate}^* \quad \text{End(FaultHandler)}
\]

The match of the fault name and variable is done with respect to the following rules [13]:

- A catch defining `faultName` only matches faults with `faultName`.
- A catch with `faultVariable` only, matches faults with `faultVariable` and any `faultName`.
- A catch defining both `faultName` and `faultVariable` matches faults with the respective `faultName` and `faultVariable`, and
- A catchAll matches all faults.

Consequently, in order to properly match the Catch pattern to be executed in case of a fault, we assign levels to catch activities, as follows:

1. For catch activities that define both `faultName` and `faultVariable` attributes, we consider a level 1,
2. For catch activities that define only `faultVariable` attributes, we consider a level 2, and

\(^{10}\)Note that in order to simplify the translation, we do not treat differently the termination of the BPEL activities, as the BPEL semantics of activity termination [13] notes. Instead, we simply clear all tokens of the pattern corresponding to the activity enclosed by the scope.
3. For *catch* activities that only define *faultName* attributes, we consider a *level 3*.

When a fault is received by the *FaultHandler*, the *BeginFaultHandler* task of *Begin(FaultHandler)* is in charge of deciding which *Catch* pattern has to be selected for execution (if any). Note that *BeginFaultHandler* (see Figure 5.19) is a composite task that links sequentially dummies corresponding to all *level 1*, *level 2*, or *level 3* *Catches* in lexical order of appearance in the input BPEL file.

Each dummy inputs *myFaultName* and *myFaultVar* variables that correspond to

---

**Figure 5.18:** The *FaultHandler* pattern template.

**Figure 5.19:** The *BeginFaultHandler* task of *Begin(FaultHandler)*.
the faultName and faultVariable attributes defined by the corresponding catch in the business process to be translated, as well as fName and fVar global (viz., EWF-net) variables that carry the fault information and which are to be set by patterns generating errors (e.g., Throw, Receive). Note that for faults without data, we assume the fVar is set to UNDEF. Moreover, each dummy also inputs the level of the Catch (viz., myLevel) and the level of the currently selected Catch (viz., currentLevel), as well as its id (viz., myId, uniquely assigned by the translator in ascending order with respect to the lexical order of the catches in the fault handler), and the id of the currently selected Catch (viz., selectedId). The currentLevel and selectedId variables are initialised by the first task of BeginFaultHandler. Furthermore, each dummy catch task sets the currentLevel and selectedId variables as indicated by the following pseudocode:

If myLevel < currentLevel then

    If myLevel = 1 and myFaultName = fName and myFaultVar = fVar then
    // viz., a catch with faultName and faultVariable that
    // matches hence it is a candidate to process the fault
    currentLevel = myLevel
    selectedId = myId
    Else
    If myLevel = 2 and myFaultVar = fVar then
    // viz., a catch with faultVariable only that
    // matches hence it is a candidate to process the fault
    currentLevel = myLevel
    selectedId = myId
    Else
    If myFaultName = fName and fVar = UNDEF then
    // viz., a catch with faultName only that
    // matches hence it is a candidate to process the fault
    currentLevel = myLevel
    selectedId = myId

A fault received by the Begin(FaultHandler) pattern is passed on to the first Catch pattern. Note that the patterns corresponding to the catch/catchAll activities are linked in a sequence (see Figure 5.18) so as to achieve dead-path-elimination inside the FaultHandler both when a Catch does not match the fault, and after the fault is processed by a Catch or CatchAll pattern (see next).

Each Pattern Template that corresponds to a catch activity (see Figure 5.20) has a guard condition checking whether the respective Catch pattern has to be skipped
or executed. The **Catch** is executed only if the **parentSkip** variable is *false* and the fault was not already processed (viz., so as to cope with identical **Catch**es), and if the **Catch** was selected for execution by the **Begin(FaultHandler)** pattern. Otherwise, the **Catch** is skipped. Note that unprocessed faults (viz., $FAULT = \text{YES}$) reaching a **CatchAll** pattern automatically lead to the execution of the **CatchAll**, provided the **parentSkip** is *false*.

Furthermore, for **Catch/CatchAll** patterns the **Success** task (or the **ActivitySpecificTask** if the **Success** task is not defined) has to set the $FAULT$ variable to $\text{ok}$ as the fault has been caught and processed successfully.\footnote{The modifications brought here to the pattern template of a **catch/catchAll** activity are illustrated on the **Basic Pattern Template**, which has to be used if the respective activity is a basic one (with the exception of the BPEL **assign** and **compensate**). Otherwise, the respective modifications are to be made to the **Begin** and/or **End** patterns of the corresponding **Pattern Template** translating the structured activity.}

Please note as well, that an error generated by an activity inside a BPEL **catch/catchAll** of a **fault handler** has to be signalled to the **fault handler** of the enclosing scope (or process), and hence the red output of the activity’s pattern should be connected to the **Begin(FaultHandler)** of the parent scope of this current scope (or of the **Process** pattern if no parent scope exists).

Furthermore, **Begin(FaultHandler)** uses a **RedGate** (instead of a **BlueGate**) that waits for red tokens to be sent (viz., faults to be raised) from inside the **Pattern Template** (or from inside the **EventHandler** of the scope’s activity). In order to interrupt the normal execution of the scope in case of a fault being raised, the **RedGate** uses a cancellation set that includes all patterns of the **Pattern Template** translating the scope’s activity and **EventHandler** except the **CompensationHandler** patterns corresponding to scopes nested in its scope.

\[\textit{Figure 5.20: The Pattern Template translating a catch/catchAll activity.}\]
Begin(FaultHandler) inputs three green lines (see Figure 5.18): (1) from Begin-Scope, (2) from End(EventHandler), and (3) from the Pattern Template translating the scope’s activity. Furthermore, its successful execution sets the FAULT variable to yes, as a fault has been raised. It is important to note that if the Scope is skipped, then Begin(Scope) sends a green (skipping) token to Begin(FaultHandler) (see gi₁ in Figure 5.18). Still, two more green tokens can arrive at Begin(FaultHandler) (see gi₂ and gi₃ in Figure 5.18) from the End(EventHandler) (if any) and from the Pattern Template of the scope’s activity. This last two redundant green tokens are to be cancelled finally by End(Scope). If the scope activity terminates (viz., the scopeActEnded variable is set to true), the FaultHandler is skipped and dead-path-elimination is employed inside it.

Finally, if the BPEL process does not define a fault handler, the translator generates a default FaultHandler pattern consisting of Begin(FaultHandler) and End(Fault-Handler) only, linked in a sequence. In this way, the faults received by this default FaultHandler will be forwarded (through EndScope) to the FaultHandler of the parent scope (or the one associated to the entire business process).

Please note that Begin(FaultHandler) may output blue tokens (Figure 5.18). For each activity X directly included inside the scope corresponding to this fault handler that has an outgoing synchronisation link bo_k that crosses the boundary of this scope and targets activity Y, we consider an identical synchronisation link bo_k emerging at the Begin(FaultHandler) and targeting Y. We do so in order to cope with scenarios in which either 1) a fault is raised in the scope prior to the execution of X, or 2) the execution of X is erroneous.

Note that a fault in the scope leads to the execution of its fault handler and to interrupting the execution of all running activities inside the scope. The former is achieved by sending a red token from the erroneous pattern to the Begin(FaultHandler) pattern, while the latter is achieved through a cancellation set that covers the (possibly structured) pattern corresponding to the scope’s activity.

When Begin(FaultHandler) receives a fault, the BeginFaultHandler tasks sets the FAULT variable to yes. Then, the ComputeTransitionConditions computes the statuses of the bo_k links as follows: bo_k = act_X.completed and bo_k. If the pattern corresponding to activity X was successfully executed (viz., act_X.completed = true) then the Begin(FaultHandler) pattern does not change the status of the corresponding bo_k link. Otherwise, it sets it to false. Note that in case of a fault (viz., FAULT = yes) Begin(FaultHandler) outputs blue tokens also on the links corresponding to the patterns completed successfully because, the target activity Y may have other incoming links from activities Z inside the same scope such that Z were not executed prior to the generation of the fault.

Last but not least, note that activities Y that have at least one incoming link emerging from an inner scope should define two additional tasks, call them Blue Gate 1 and Blue Gate 2. In this case, the join of the Blue Gate task of Y should be a XOR instead of a standard AND, and it should input the other two Blue Gate 1
5.2. FROM BPEL TO YAWL

Figure 5.21: The EventHandler pattern template.

and Blue Gate 2 tasks. Now, Blue Gate 1 serves as a normal Blue Gate for Y, and it should input the incoming blue links of Y, while Blue Gate 2 should input the incoming links of Y that were added by the fault handler of the inner scope. The scenario is similar if Y has incoming links from various inner scopes – it basically changes the number of dummy Blue Gate k tasks.

EventHandler

In the pattern of the event handler (Figure 5.21):

\[
\text{Begin(EventHandler)} \quad \text{PatternTemplate}^+ \quad \text{End(EventHandler)}
\]

the Pattern Templates execute concurrently. On the one hand, the patterns of onMessage activities are placed in a loop with a guard that checks the end of the Pattern Template translating the activity inside the scope. On the other hand, the patterns of onAlarm activities are executed at most once as an alarm event is carried out at most once while the corresponding scope is active.

The Begin(EventHandler) pattern template of the outermost scope (or of the Process pattern) (Figure 5.22) has to wait for a green (enabling) token from a Receive or Pick onMessage, whose createInstance attribute is set to yes. Furthermore, Begin(EventHandler) outputs green tokens for all the onMessage and onAlarm patterns in order to enable them.
CHAPTER 5. TRANSLATING BPEL PROCESSES INTO YAWL WORKFLOWS

Figure 5.22: The Begin(EventHandler) pattern template.

Each onMessage pattern (bottom-left part of Figure 5.21) is composed by a Receive-like pattern and by the actual Pattern Template translating the activity defined in the BPEL onMessage. In order to execute the Receive pattern either after the Begin(EventHandler) (viz., the first time the respective onMessage is executed), or after the Pattern Template (viz., successive executions of the respective onMessage pattern), its GreenGate employs a XOR-join. Both patterns of the onMessage have to check whether the scope’s activity has ended (viz., scopeActEnded is true). On the one hand, the Receive does the check in its GreenGate task. If the scope’s activity ended before the Receive, then both patterns are skipped. On the other hand, the Pattern Template does the check when outputting green tokens (see the respective predicate in Figure 5.21). We do so because the scope’s activity may have finished after executing the Receive, and in this case, events that are running are allowed to finish their execution.

All onAlarm patterns (bottom-right part of Figure 5.21) are enabled by the Begin(EventHandler) pattern and they can be executed at most once. Each one consists of a Wait, linked in a sequence with a Pattern Template. The Wait implements the timer and if it finishes its execution, the Pattern Template translating the onAlarm activity gets executed. Note that the pattern of the scope’s activity cancels the Wait timer and (in order not to lock the workflow) it forwards a green (skipping) token to the onAlarm Pattern Template. Similarly to a Receive for the onMessage pattern, the Pattern Template here employs a XOR-join for its GreenGate. However, the decision on whether to execute the Pattern Template is based on a FinishedK variable (whose initial none value is given by the Begin(EventHandler) pattern, and further set to executed by the ComputeTransitionConditions of the Wait timer), and not on the the status of the scopeActEnded variable. We do so because the Pattern Template should be executed if and only if the Wait timer was executed successfully. Otherwise, it should be skipped.
5.2. FROM BPEL TO YAWL

The CompensationHandler pattern template.

CompensationHandler

Finally, the CompensationHandler pattern (Figure 5.23) consists of:

\[ \text{Begin(CompensationHandler)} \] \text{PatternTemplate} \ast \text{End(CompensationHandler)} \]

If the scope completes successfully, the Begin(CompensationHandler) is activated and waits for a green token from a Begin(Compensate) pattern (see next). The green output of Begin(CompensationHandler) enables the Pattern Template implementing the actual compensating activity, which forwards (on its termination) the green token to End(CompensationHandler). Upon completion, the End(CompensationHandler) issues only one green token on one of its two green outputs. If the Scope is skipped, End(CompensationHandler) sends a green token to the pattern translating the activity (structurally) following the scope in the BPEL process. Otherwise, it sends a green token to the End(Compensate) pattern (see next). Note that if a BPEL scope does not define a compensation handler yet there is a compensate activity targeting the respective scope, the translator generates a default CompensationHandler consisting only of Begin(CompensationHandler) directly linked to End(CompensationHandler).
Compensate

The BPEL *compensate* has the following structure:

```xml
<compensate scope="ncname"? standard-attributes>
  standard-elements
</compensate>
```

The *compensate* activity serves for triggering roll-back activities as a result of e.g., faults occurring in the business process. We recall that specifying the name of the scope to be compensated leads to invoking the *compensation handler* of the respective *scope*. Otherwise, the default compensation mechanism is triggered, and it (roughly) involves the invocation of the the compensation handlers for the immediately enclosed scopes in the reverse order of the completion of these scopes.  

The pattern template corresponding to the BPEL *compensate* is given in Figure 5.24:

```
Begin(Compensate)   End(Compensate)
```

since *compensate* terminates only when the invoked *CompensationHandler* finishes its execution. Recall that we consider only explicit compensation, that is *compensate* activities specifying the name of the scope to be compensated, and furthermore, without considering *scopes* nested inside *while* activities. *Begin(Compensate)* sends a green token directly to *End(Compensate)* if the *compensate* is skipped, or if the scope to be compensated did not finish its execution. Otherwise, the green token is sent to the *Begin(CompensationHandler)* of the scope to be compensated. Dually, *End(Compensate)* receives a green token either directly from *Begin(Compensate)*, or from the *End(CompensationHandler)* of the scope to be compensated. Then, it forwards it to the pattern structurally following the *Compensate*.

### 5.2.3 BPEL Processes

A BPEL *process* encapsulates the process activity and it can further define a *fault handler*, a *compensation handler*, as well as an *event handler*, similarly to a *scope*. Its structure is the following:

```xml
<process name="ncname" targetNamespace="uri"
  queryLanguage="anyURI"?
  expressionLanguage="anyURI"?
  suppressJoinFailure="yes|no"?
  enableInstanceCompensation="yes|no"?
  abstractProcess="yes|no"?
  xmlns="http://schemas.xmlsoap.org/ws/2003/03/business-process/"/>
```

---

12Note that a *compensate* may only be used inside the *fault handler* or the *compensation handler* of the scope that immediately encloses the scope to be compensated [13].

13Note that each *Compensate* should have an identification so that the *End(CompensationHandler)* can know which *Compensate* pattern invoked it.
5.2. FROM BPEL TO YAWL

The Compensate Pattern Template

Compute Transition Conditions

Green Gate

parentSkip

go

Begin (Compensate)

p1

gi

gi1

parentSkip = F

parentSkip = T

Blue Gate

skip

joinCondition

scopeName

joinCondition

suppressJoinFailure

fault = F

fault = T

End (Compensate)

Compensate

where:

(1) go1 goes to End(Compensate), and p1 is skip = T or ScopeActEnded(scopeName) = F, and

(2) go2 goes to Begin(CompensationHandler) of the Scope named scopeName, and p2 is not p1.

Figure 5.24: The Compensate pattern template.
Differently from the *scope*, a *process* may define *partner Links* and *partners* of the business process. The former represents a conversational relationship between two partner processes, while the latter represents the capabilities of a partner service as a subset of the partner links of the process. Furthermore, faults that reach the (possibly default) *fault handlers* lead to an abnormal termination of the business process (similarly to a *terminate*), even if they are processed successfully. Since the *compensation handler* is installed only after the successful termination of the activity defined by the process, a business process instance can be compensated only by platform-specific means. Note that in order to allow such compensation the *enableInstanceCompensation* process attribute has to be set to *yes*.

The *Process* pattern (Figure 5.25):

```
Begin(Process) FaultHandler [EventHandler] PatternTemplate End(Process)
```

resembles very much the *Scope* pattern, although there are some differences between the two, presented hereafter. For example, *Begin(Process)* and *End(Process)* have to be connected to the *input condition* and to the *output condition*, respectively, of the workflow. *Begin(Process)* enables the *Pattern Template*, the *FaultHandler*, as well as the *EventHandler* (if any), and it is in charge of setting the initial *false* values of the *somebodyCreatedAnInstance* and *faultyProcess* variables. The former is set to *true* by the execution of a *Receive* or of a *Pick onMessage* with a *createInstance* variable having a *yes* value, while the latter is set to *true* if a fault is caught by the process *FaultHandler*. Due to the fact that the BPEL *process* cannot have a parent activity, the *GreenGate* task of its *Begin(Process)* pattern simply inputs an always *true parentSkip* variable. Dually, as the activity defined by the process cannot be skipped, *Begin(Process)* outputs an always *true parentSkip*.

If the BPEL process does not define a *fault handler*, or if it does but it does not contain a *catchAll* clause, one (default) *FaultHandler* with a default *catchAll* (viz.,
an *Empty* pattern) must be defined in the *Process* pattern. This is needed to catch all uncaught faults being raised within the process. Note that the reception of a fault by the process *FaultHandler* leads to an abnormal process termination, even if the fault is processed. Furthermore, faults being raised (and uncaught) inside the process *FaultHandler* lead to the immediate execution of the *End(Process)* pattern, as in the case of a *Terminate* (see next).

The *EventHandler* is active for the entire process lifetime and the *Pattern Template* of the activity defined by the process is in charge of clearing its tokens upon its completion, similarly to a *Scope*. In order to minimise the number of cancellation sets defined in the workflow, all *Terminate* patterns forward the green token to *End(Process)*, which is in charge of immediately terminating the entire business process. It does so by clearing all the tokens of the *Pattern Template* corresponding to the activity defined by the process. Hence, *End(Process)* is enabled if it receives either a green token from a *Terminate*, or from the process *FaultHandler*.

As previously mentioned, the *compensation handler* can only be invoked by platform-specific means. However, we do not consider a *CompensationHandler* pattern for the entire business process, since YAWL does not allow for such invocation mechanism. (Note also that the *CompensationHandler* pattern of the process would block the workflow waiting for a green token.)

A BPEL process is translated into a YAWL workflow by instantiating the *Process* pattern. This leads to recursively instantiating the *Begin(Process)*, *FaultHandler*, *EventHandler* (if any), and *End(Process)* patterns, as well as the *Pattern Template* corresponding to the activity defined by the BPEL process. Note that the instantiation of a pattern takes into account the context in which the activity is placed inside the BPEL process. Namely, instantiating a pattern means adjusting the (number of) input and output lines, setting and mapping the inputs and outputs of the tasks.
in the pattern, as well as suitably interconnecting its child patterns. The instantiating process bottoms-out at \textit{basic pattern templates}. More information on how to instantiate the pattern templates is given in the next Section, which illustrates the YAWL workflow one obtains by translating a simple BPEL process.

5.3 Example: Complete Translation of a (Simple) BPEL Process

In this Section we describe the translation of the Greatest Common Divisor (GCD) BPEL process introduced in Section 5.1. We recall that the GCD process computes the greatest common divisor of two numbers by repeatedly raising an exception while one of the two numbers is bigger than the other and by decreasing its value in the corresponding catch.

In the following, we first describe in detail the generation of the YAWL workflow translating the GCD process (Subsection 5.3.1), followed by an execution scenario of the obtained GCD workflow (Subsection 5.3.2).

5.3.1 Generating the GCD Workflow

Figure 5.26 gives the high-level view of the YAWL workflow obtained from the GCD process, while a more detailed view of the GCD workflow is presented in Figure 5.27.\(^{14}\)

Roughly, the BPEL2YAWL translator inputs the GCD process, it parses it and produces the corresponding GCD YAWL workflow as described hereafter. However, please note that for space issues we cannot depict each step of the translation in a separate figure. Furthermore, we shall not give an in-depth description of the pattern instances (e.g., mapping of all the task variables) in order to keep the discussion comprehensible.

\textbf{Step 1: Instantiating the Process Pattern Template.}

The translation starts with a GCD workflow consisting only of the input and output conditions. Initially, the translator generates the \texttt{Begin(Process)} and \texttt{End(Process)} patterns, and it suitably connects them to the input and output conditions of the GCD workflow (see Figure 5.26, top-left and top-right, respectively).

On the one hand, \texttt{Begin(Process)} sets the global (viz., EWF-net) variable \texttt{suppressJoinFailure} to \texttt{yes}, as well as it defines and sets the \texttt{name} input variable of its \texttt{BeginProcess} task to \texttt{GCD}, as defined in the \texttt{<process>} element of the BPEL process. Furthermore, it also maps a \texttt{true} boolean value into a (first) \texttt{parentSkip} EWF-net variable, which is to be inputted by both patterns translating the activity

\footnote{The full BPEL process and the YAWL workflow of the example can be downloaded from: \url{http://www.di.unipi.it/~popescu/GCD_Example.zip}.}
5.3. EXAMPLE: COMPLETE TRANSLATION OF A (SIMPLE) BPEL PROCESS

and the fault handler of the business process. (Although we shall not refer to indexes when discussing about e.g., parentSkip variables, the readers should note that each pattern instance that translates the beginning of a structured activity outputs a new parentSkip variable, which is to be inputted by the rest of the patterns translating the respective structured activity.) Begin(Process) is represented in Figure 5.27 (top-left) by the GreenGate atomic tasks and by the BeginProcess composite task. Note that the latter is in charge of mapping the IOs previously mentioned.

On the other hand, End(Process) is instantiated by default, that is, to an Empty-like pattern, without any significant inputs and/or outputs. A more detailed view of End(Process) reveals the GreenGate as well as the EndProcess tasks in Figure 5.27 (top-right).

BPEL2YAWL continues next by (recursively) translating the fault handler as well as the flow activity of the GCD process. As we shall see next, these two patterns are enabled by the green outputs of Begin(Process).

Step 2: Instantiating the Process’ FaultHandler.

The fault handler of the GCD process defines a catch that processes negNum faults. The activity of the catch is a reply, which forwards to the invoker of the business process the respective fault. Consequently, BPEL2YAWL generates a FaultHandler pattern instance consisting of a Begin(FaultHandler), End(FaultHandler), as well as one Reply and one Empty patterns, all linked sequentially by green lines as seen in Figure 5.26 (bottom-left).

In Figure 5.27 (centre-left) one may see that Begin(FaultHandler) consists of the GreenGate, RedGate2, RedGate1, GreenGate0, and BeginFaultHandler tasks. The GreenGate task inputs the green output of the Begin(Process), as the initiation of the business process structurally enables the fault handler. Furthermore, as we shall see later, RedGate1 serves for catching red tokens representing faults raised in the process (e.g., by the throw activity in the flow), while GreenGate0 serves for inputting the green token of End(Flow), which enables and skips the FaultHandler if the process’ activity (viz., the flow) completes successfully. Another characteristic of the RedGate1 task consists of its cancellation set that will include the entire Flow pattern. The purpose of this cancellation set is to interrupt the execution of the activities inside the flow when the process’ fault handler receives an error.

Furthermore, BPEL2YAWL generates another cancellation set associated to the End(Process) pattern, which includes the RedGate2, RedGate1, and GreenGate0 tasks of the Begin(FaultHandler) pattern. We recall that this is useful in order to cancel redundant red tokens if e.g., multiple failures reach Begin(FaultHandler) prior to its execution, as well as redundant green tokens due to skipping the Scope. For simplicity, we represent this cancellation set in Figure 5.26 as including the entire Begin(FaultHandler) pattern. However, we do not represent the cancellation set in Figure 5.26 in order not to burden further the workflow. Please note that, in order to keep the translation manageable, we do not differentiate between activities that
should be allowed to complete (e.g., assigns) and activities that are to be interrupted. We simply interrupt all running activities by removing all tokens of the pattern translating the scope’s activity.

Begin(FaultHandler) forwards a green token to the Reply pattern, represented in Figure 5.26 by the GreenGate and Reply tasks. Since the reply in this scenario is a catch activity, the GreenGate of its Reply pattern is guarded by the fault = negNum boolean expression (see the G2 comment in Figure 5.27). (In order to ease the presentation we refer here and in the Figures to the “core” of the catch guard that also takes into account the parentSkip and FAULT variables, as defined in Section 5.2.) As a consequence, at run-time of the GCD workflow, if the process’ fault handler receives a fault due to e.g., a message mismatch in the receive, the guard of the Reply pattern will evaluate to false, and hence the Reply will be skipped. However, as the fault handler of a BPEL business process has to catch all unprocessed faults in the process, BPEL2YAWL generates the pattern of a default catchAll represented by the Empty pattern immediately following the Reply. Note that, in order to suppress all uncaught faults, the GreenGate of the Empty pattern simply does not employ a guard testing the match with the faultName variable. Finally, the Empty pattern forwards a green token to End(FaultHandler), which similarly to End(Scope) is instantiated by default.

As indicated in Figure 5.27 (centre-right), End(FaultHandler) outputs a green token for the GreenGate of the End(Process) pattern. Roughly, we recall that the FaultHandler is either executed in case of a failure, or skipped if the process’ activity completes successfully, and in both such cases the FaultHandler is immediately followed by the termination of the business process.

Step 3: Instantiating the Process’ Flow.

The activity defined by the GCD process is a flow and, as a result, the translator generates instances of the Begin(Flow) and End(Flow) patterns, and suitably links them to the Begin(Process) (Figure 5.26 and 5.27, top-left) and Begin(FaultHandler) (Figure 5.26 and 5.27, top-right), respectively. Furthermore, both patterns are instantiated by default, as the flow activity is included in a “simple context” (i.e., it is not subject to any guard, or it does not have any incoming or outgoing links).

The flow consists of four activities – a receive, a throw, a while, and a sequence. The Receive pattern instance generated by BPEL2YAWL is composed of a GreenGate task and of a Receive task (see Figure 5.27, top-centre). The former structurally enables the Receive task (when the Flow is executed) by inputting the green output of Begin(Flow), while the latter forwards a green token to End(Flow) on its completion. Furthermore, the translator generates for the Receive a red output line that targets the RedGate1 task of the process Begin(FaultHandler) pattern, necessary for signalling faults raised by e.g., mismatches in the input message of the receive. (We recall that such a failure is generated by the execution of the Fault task of the Receive composite task.)
The second activity in the flow is a *throw* which results in the *Throw* pattern given in the top-centre part of Figure 5.27. Similarly to the *Receive* it employs a *GreenGate* task and it is connected by green lines to *Begin(Flow)* and *End(Flow)*. However, it also defines a *BlueGate* as the *throw* activity in the BPEL process is target of a synchronisation link having the *receive* as source. Consequently, the translator generates a blue line that suitably links the *Receive* composite task to the *BlueGate* of the *Throw* pattern. Moreover, the YAWL (boolean) predicate of the blue link is given by the (boolean) transition condition defined by the respective synchronisation link in the GCD process (see the TC2 annotation in Figure 5.27). Furthermore, since the activity being translated is a *throw*, BPEL2YAWL adds a red line linking the *Throw* composite task of the *Throw* pattern to the *RedGate1* task of the *Begin(FaultHandler)* pattern (belonging to the process’ *FaultHandler*).

Through this red line the *Throw* pattern interrupts the normal execution of the GCD workflow if one of the two numbers inputted by the *Receive* is negative or zero. (Note that the *Throw* pattern cannot raise a *joinFailure* since, even if it has to be skipped when one of the two inputted numbers is negative or zero, the corresponding *suppressJoinFailure* variable is set to *YES* for the whole process.)

Now, because the remaining two activities of the *flow* are structured ones, we shall describe them in further separate steps.

### Step 4: Instantiating the While Pattern Template.

The *while* activity initially leads to the generation of the *Begin(While)* (Figure 5.26 centre-left) and *End(While)* (Figure 5.26 centre-right) patterns. As shown in the centre-left part of Figure 5.27, *Begin(While)* consists of three tasks – a *GreenGate*, a *BlueGate*, as well as a *BeginWhile*. The *GreenGate* structurally enables the *Begin(While)* pattern by inputting the green output of *Begin(Flow)*. The *BlueGate* task inputs the blue output of *Receive* and, at run-time, it decides whether to skip or to execute the *While* pattern. If both numbers inputted by the *Receive* are strictly positive (viz., the TC1 predicate in Figure 5.27 evaluates to *true*), the *Receive* task outputs a blue token that leads to the execution of the *While*. Otherwise, the blue token leads to skipping the *While*. Apart from the green inputs given by the *GreenGate* and *BlueGate*, the *BeginWhile* task inputs one more green tokens from *End(While)*, which serves for re-cycling. Moreover, it employs one green output that enables its *scope* child activity. Furthermore, the *ExecOrSkip* task of *BeginWhile* employs a guard (corresponding to the guard of the *while* activity in the GCD process) that checks whether the two numbers inputted by the *Receive* are equal. At run-time, if the guard holds true the *While* pattern will be skipped, otherwise it will be executed.

The *End(While)* pattern (Figure 5.27 centre-right) is instantiated by default. It has a *GreenGate* that will have to input the green output of the pattern translating the *scope* (child) activity of the *while*, as well as an *EndWhile* task that outputs two green tokens – one for the *BeginWhile* task (guarded by the TC3 boolean predicate in Figure 5.27), and another for the *GreenGate* of the *End(Flow)* pattern (guarded
Step 5: Instantiating the Scope Pattern Template.

The scope encloses a switch activity and it defines a fault handler as well. Consequently, BPEL2YAWL creates instances of the Begin(Scope) and End(Scope) patterns, and it suitably links them to Begin(While) (Figure 5.26, centre-left) and End(While) (Figure 5.26, centre-right), respectively.

On the one hand, Begin(Scope) consists of a GreenGate task that enables the Scope, as well as of a BeginScope task that will have to enable both the Switch and scope’s FaultHandler pattern instances (Figure 5.27).

On the other hand, End(Scope) employs a GreenGate that has to wait for the green token from the scope’s FaultHandler, and an EndScope task that will forward it to the GreenGate of the EndWhile pattern. Furthermore, EndScope outputs a red line that targets the RedGate1 task of the process Begin(FaultHandler) in order to forward to it exceptions caught yet unprocessed by the scope FaultHandler.

BPEL2YAWL continues next with the translation of the scope’s fault handler and of the switch activity.

Step 6: Instantiating the Scope’s FaultHandler Pattern Template.

The fault handler consists of two catches each one wrapping an assign activity. As a result, the translator instantiates a Begin(FaultHandler), two Assign patterns, as well as an End(FaultHandler), all linked in a sequence.

Begin(FaultHandler) is similar to the Begin(FaultHandler) pattern of the process’ FaultHandler (Figure 5.27 bottom-left). The GreenGate task inputs the green token of BeginScope, while the RedGate1 and GreenGate0 input red and green tokens, respectively, of the Switch pattern enclosed in the Scope. The role of the BeginFaultHandler is to enable the pattern of the first Catch in the FaultHandler. Moreover, its RedGate1 task defines a cancellation set in charge of interrupting the Switch pattern of the Scope.

Dually, EndFaultHandler has a GreenGate that receives the green token from the last (viz., second) Catch in the FaultHandler, as well as an EndFaultHandler task that has to send the green token to the GreenGate of the End(Scope) pattern.

BPEL2YAWL also generates a cancellation set associated to the End(Scope) pattern, which includes the RedGate2, RedGate1, and GreenGate0 tasks of the scope’s Begin(FaultHandler) so as to cancel redundant red and green tokens that might get stuck due to e.g., multiple (simultaneous) failures, or to skipping the Scope pattern. (For simplicity, the cancellation set is represented in Figure 5.26 as including the entire Begin(FaultHandler) pattern.)

Next, both catches translate to Assign patterns composed of Begin(Assign), Copy, and End(Assign) pattern instances (Figure 5.26, bottom-right). A main characteristic of the two is that Begin(Assign) employs a GreenGate task that checks the catch
guard. Consequently, the first Assign is executed if the fault name is \( \text{dec}_a \) (see the G6 comment in Figure 5.27, bottom-centre), while the second Assign is executed provided the fault name is \( \text{dec}_b \) (see the G7 comment in Figure 5.27, bottom-centre). Otherwise, the respective Assigns are skipped. As indicated in the GCD process, the Copy task of the first Copy pattern maps the expression \( a - b \) into the variable \( a \), and similarly, the Copy task of the second Copy pattern maps the expression \( b - a \) into the variable \( b \). Furthermore, BPEL2YAWL adds for each of the two Copy tasks a red output linking them to the RedGate1 task of the process’ Begin(FaultHandler) in order to forward to it (possible) faults due to assignment issues (e.g., parameter types mismatches).

The translation continues next with the switch activity of the scope.

**Step 7: Instantiating the Scope’s Switch Pattern Template.**

The switch is translated into a Begin(Switch), two Throws, as well as an End(Switch) pattern, all linked sequentially (Figure 5.26, centre). Begin(Switch) and End(Switch) are instantiated by default, and include GreenGates that enable the BeginSwitch and EndSwitch tasks, respectively (Figure 5.27, centre). Moreover, the former is enabled at the beginning of the Scope, while the latter enables the scope’s FaultHandler upon (successful) completion of the Switch.

Now, since the first throw is a case branch, its Throw pattern defines a GreenGate that checks whether a previous branch was already executed, as well as the branch guard, as defined in the GCD process. As a consequence, the first Throw task is executed if and only if \( a > b \) (see the G4 comment in Figure 5.26), while the second one (corresponding to the otherwise branch) is executed otherwise. Moreover, both Throws output a red line that signal \( \text{dec}_a \) and \( \text{dec}_b \) faults, respectively, to the FaultHandler of the Scope.

Finally, BPEL2YAWL terminates by translating the sequence activity of the flow.

**Step 8: Instantiating the Flow’s Sequence Pattern Template.**

The sequence defines two activities – an assign followed by a reply, and hence it leads to the generation of a Begin(Sequence), an Assign, a Reply, as well as an End(Sequence) pattern (Figure 5.26 bottom).

Begin(Sequence) (Figure 5.27 top-left) defines a GreenGate that receives a green token from the BeginFlow task and which serves for structurally enabling the Sequence, as well as a BlueGate that is the target of a blue line from End(While) due to the synchronisation link between the while and the sequence in the GCD process. Consequently, although Begin(Sequence) is always enabled when it receives both the green and blue tokens, it will be executed only when the status of the synchronisation link is positive (viz., the joinCondition computed by the BlueGate, which is given by the BPEL transitionCondition, holds true).
Since the synchronisation link does not define a transitionCondition, BPEL assumes it to be true by default, and hence BPEL2YAWL considers an (always) true value for it in the BlueGate of Begin(Sequence), for the computation of the joinCondition. (We recall that the transition conditions do not translate to YAWL predicates, and hence blue tokens are outputted on blue lines even if their corresponding transition conditions are false (viz., they have negative statuses). However, as just mentioned, each transition condition is mapped onto a EWF-net variable by the source pattern of the link, and it is taken into account by the BlueGate of the target pattern when computing the joinCondition.

It is important to note that, although the sequence is target of a synchronisation link, the BeginSequence task of Begin(Sequence) does not output a red line for the RedGate1 task of the process’ Begin(FaultHandler) pattern, since its corresponding suppressJoinFailure is set to yes and hence, it cannot raise joinFailures.

On the other hand, the End(Sequence) pattern (Figure 5.27 top-right) is instantiated by default; it simply waits for the completion of the Reply pattern, and it forwards the green token to the GreenGate of the End(Flow) pattern.

The Assign (Figure 5.27 top-centre) is constructed similarly to the previous ones defined in the FaultHandler of the Scope (yet in this case there is no boolean guard constraining the execution of the Assign). The Copy pattern maps the (input) variable a into an (output) variable c, as given by the respective copy tag in the BPEL process. Furthermore, since the mapping might raise faults, the Copy task outputs a red line that targets the RedGate1 task of the process’ Begin(FaultHandler) pattern.

Finally, the translator produces an instance of the Reply pattern, consisting of a GreenGate and of a Reply composite task (Figure 5.27 top-right), both instantiated by default and linked in the workflow correspondingly. Since the reply activity may raise errors, the Reply task is linked to the RedGate1 task of the process’ Begin(FaultHandler) pattern through a red line.
5.3. EXAMPLE: COMPLETE TRANSLATION OF A (SIMPLE) BPEL PROCESS

Figure 5.26: High-level view of the YAWL workflow translating the GCD BPEL process.
Figure 5.27: Detailed view of the YAWL workflow translating the GCD BPEL process.
5.3.2 Use Case of the GCD Workflow

Consider now an execution scenario in which the two input variables \( a \) and \( b \) – take the values of 2 and 4, respectively. In the following we shall describe the step-by-step execution of the GCD workflow by referring to its high-level view given in Figure 5.26.

The workflow executes first \texttt{Begin(Process)} (that outputs two green tokens) followed by \texttt{Begin(Flow)} (that outputs four green tokens) and by \texttt{Receive} (that outputs one green token). As both numbers are strictly positive, \texttt{Receive} sends a blue token to \texttt{Begin(While)} and another blue (skipping) token to \texttt{Throw}. Because the \texttt{suppressJoin-Failure} (set for the entire process only) has a \texttt{yes} value, skipping the \texttt{Throw} does not raise a \texttt{joinFailure}, but forwards the green token to \texttt{End(Flow)}.

The execution continues with \texttt{Begin(While)} and then with \texttt{Begin(Scope)} because \( a \neq b \) (viz., \( 2 \neq 4 \)). Then, \texttt{Begin(Scope)} forwards a green token to \texttt{Begin(Switch)} and another one to the \texttt{Begin(FaultHandler)} of the scope. The first \texttt{Throw} in the \texttt{Switch} is skipped as \( a < b \) (viz., \( 2 < 4 \)), yet the second one (of the \texttt{otherwise} branch) is executed, and a \texttt{dec\_b} fault is raised. As a result, only a red token is sent further to the \texttt{Begin(FaultHandler)} of the scope.

The first \texttt{Assign} in the scope’s \texttt{FaultHandler} is skipped (as \texttt{fault=“dec\_b”}), while the second \texttt{Assign} decreases the value of \( b \) by \( a \). Hence, \( a = 2 \) and \( b = 2 \) now. The green token will reach next \texttt{End(FaultHandler)} and then \texttt{End(Scope)} that forwards the green token to \texttt{End(While)} (as the fault was processed).

Because \( a = b = 2 \), \texttt{End(While)} sends a green token to \texttt{End(Flow)} and a blue token to \texttt{Begin(Sequence)}, which enables the \texttt{sequence}. The execution of the \texttt{Assign} inside the \texttt{Sequence} leads to copying the value of \( a \) into \( c \) (viz., \( c = 2 \)) and to replying with the latter to the client of the GCD workflow.

Finally, \texttt{End(Sequence)} outputs a green token that enables \texttt{End(Flow)}, which has now gathered all its input (green) tokens. \texttt{End(Flow)} forwards a green token to \texttt{End(Process)} that sends the green token to the output condition, marking in this way the end of the GCD workflow.

5.4 Related Work

Currently there are several approaches that tackle the translation of BPEL processes into other languages or formalisms. Moreover, most of these approaches focus on the verification of properties of business processes.

Fisteus et al. [5] describe VERBUS, a FSM-based framework for the formal verification of BPEL processes, but they do not treat synchronisation links, complex fault handling, and event and compensation handling.

Koshkina and van Breugel [51] introduce the BPE-calculus in order to formalise the control-flow of BPEL and build upon it a tool for the analysis of business processes. Still, they do not tackle fault and compensation handling.
Hinz et al. [41] give a PN semantics to BPEL processes by defining a pattern for each BPEL activity. However, they abstract from data and leave out transition guards. Consequently, control-flow decisions based on the evaluation of data are replaced by non-deterministic choices. Our approach does not suffer from this limitation as both BPEL and YAWL use XMLSchema and XPath for data manipulation, and hence the data translation between the two is straightforward.

Ouyang et al. [70] formalise BPEL in terms of PNs with the purpose of analysing its control-flow. Although they handle both synchronisation links and exceptional behaviour, their approach is focused on the analysis of business processes, and it cannot be directly exploited to compose business processes.

Camara et al. [43] propose a CCS based formalisation of BPEL processes. However, the authors consider only the core constructs and mechanisms of BPEL, leaving the inclusion of data and other BPEL features such as compensation and event handlers as future work. For example, by abstracting away data, the proposed formalisation of the BPEL switch does not take into account the branch guards.

A thorough analysis of formal BPEL models, verification techniques and tools can be found in [37]. However, similarly to the above mentioned approaches, existing models generally do not tackle the synchronisation links, or the data-flow aspects, or the exceptional behaviour of BPEL processes.

Our main concern here was the translation of BPEL processes into YAWL workflows with the purpose of contributing to the automation of the processes of Web service aggregation and adaptation.

However, it is worth noting that the translation of BPEL processes into YAWL workflows also gives the possibility of formally analysing business processes. YAWL is built on top of Petri nets, and it has a well-defined formal semantics based on transition systems, hence tools such as [90] and [91] can be employed to formally analyse YAWL workflows. Furthermore, in Section 3.4 we gave an insight on how reachability graphs and modified reachability trees can be employed to formally check properties of YAWL workflows such as, lock-freedom, liveness, and so on.

5.5 Discussion

In this Chapter we have outlined the specification of a BPEL2YAWL translator of BPEL processes into YAWL workflows. As we already anticipated at the beginning of the Chapter, the translator aims to contribute towards the semi-automated generation of service contracts from real-world service descriptions, by giving clients the possibility to automatically generate the behaviour information of the contracts.

The main strengths of BPEL2YAWL are that (1) it provides an automated pattern-based compositional translation of BPEL processes into YAWL workflows, (2) it copes with all types of BPEL activities (including flows with synchronisation links, and scopes), and (3) it handles the exceptional behaviour – events, faults and (explicit) compensation. Furthermore, (4) it straightforwardly plugs into our Web
service aggregation and adaptation methodology (described in Chapters 3 and 4),
while (5) the pattern-based compositional nature of the BPEL2YAWL translator sets
the basis for the development of an inverse YAWL2BPEL translator. Last but not
least, (6) BPEL2YAWL provides a lightweight semantics of BPEL processes, as well
as (7) it sets the basis for the formal analysis of BPEL processes.

This thesis also gives an informal validation of the translator through the trans-
formation of a few BPEL processes. In Chapter 4 we briefly described the translation
of a Mars Explorer and a Command Centre services into corresponding workflows,
which were further employed for the construction of a YAWL adapter for the two
services. Furthermore, we also gave an insight on how the respective YAWL adapter
can be deployed as a BPEL process. Moreover, in this Chapter we thoroughly de-
scribed the translation of a Greatest Common Divisor BPEL process, as well as an
execution scenario of the resulting YAWL workflow.

A Java prototype of the BPEL2YAWL translator described in this Chapter has
been implemented\textsuperscript{15}. This first version of the translator can be successfully used to
translate (simple) BPEL processes into YAWL workflows, which can be loaded and
executed into the YAWL engine\textsuperscript{16}.

In short, the prototype consists of two Java packages – BPELDoc and YAWLDoc,
for managing BPEL and YAWL documents, respectively. BPELDoc employs a data
structure that models the hierarchy of a BPEL document. For example, the BPELDoc
class has a BPELProcess object, which in turn has an Activity object, and optional FaultHandler,EventHandler, and CompensationHandler objects. Dually,
YAWLDoc uses a data structure that models the nesting of a YAWL document\textsuperscript{17}. For example, the YAWLDoc class refers to a Decomposition object, which can have multiple Task and Condition objects. The control-flow is maintained by the Tasks,
which store their input and output connections into Mapping objects.

At runtime, BPELDoc first parses the input BPEL document into a BPELDoc ob-
ject. Then, the BPELDoc object creates a YAWLDoc object, in which it suitably stores
the transformation of the BPELProcess by recursive translations, starting with the
Activity of the BPELProcess (as described in Section 5.2). Finally, YAWLDoc saves
the obtained workflow as a YAWL document. Note that YAWLDoc also gives the
possibility to load YAWL documents, so that one may use it to test the syntactic
correctness of the translated workflow. Currently, the main limitation of our imple-
mentation of the BPEL2YAWL translator is that it does not cope with complex data
structures and assignments such as mapping expressions to variables.

\textsuperscript{15}The source code of the prototype can be downloaded from:
http://www.di.unipi.it/~popescu/BPEL2YAWL.zip. Moreover, a detailed discussion of
the prototype is given in [57].

\textsuperscript{16}http://ga2377.campus.tue.nl:8080/worklist/

\textsuperscript{17}By “YAWL document” we refer to a YAWL XML file, and not to its binary representation
generated with the YAWL editor.
Chapter 6

Concluding Remarks

This thesis addresses two important open challenges of the service-oriented computing paradigm applied to Web services, namely, the aggregation and adaptation of Web services. The aim of the thesis is to contribute towards the development of Web service tools that lift the aggregation and adaptation of Web services from (mainly) manual approaches (that are error-prone and time-consuming), to semi-automated engineered processes.

Web service aggregation and adaptation are mainly hindered by the following issues:

- The lack of formal semantics of most of the Web service languages that describe service behaviour (e.g., BPEL [13], OWL-S [71], WSCI [92], WSCDL [99], WSMO [103]) does not allow for tools to be employed for the automated verification of service properties such as lock-freedom,

- The lack of ontology information of most of the Web service description languages (e.g., WSDL [100], BPEL) impedes the automated matching of service parameters, necessary for example to improve the process of discovering Web services, and the generation of data-flow dependencies among Web services.

- Furthermore, the construction of heterogeneous services is hampered by the fact that, currently, there are no tools for the automated translation between languages that describe the service behaviour.

In order to tackle such limitations, we argue that providers should expose service contracts consisting of (WSDL) signature, (YAWL) behaviour, and (OWL) ontology information.

First, the WSDL signature serves for describing the functionality of the Web service in terms of operations it offers to its invokers.

Second, we reckon that YAWL is a good candidate for expressing (part-of) the interaction behaviour of a service since, on the one hand, it provides a formal basis for the analysis of service behaviour, and on the other hand, it can be used as
a lingua-franca for representing the service behaviour and thus for the creation of heterogeneous services. Furthermore, service developers can exploit the YAWL workflows to construct the behaviour of the composite service, as well as to overcome protocol mismatches among interacting services.

Third, the OWL ontology information allows for service parameter matches (e.g., \textit{exact/plug-in/subsumes} [73]) to be automatically inferred. Such data-flow information can further be used to automatise and improve the accuracy of the service discovery process, to suitably link the workflows of services involved in a composition through data-flow dependencies, as well as to overcome ontology mismatches among the parameters of the involved services.

In short, this thesis describes a methodology for the location, aggregation, and adaptation of Web services. The methodology inputs a registry of (advertised) service contracts and a client query, and it outputs (whenever possible) services that satisfy fully or partially the client request. A client query can be expressed either functionally (as a black-box), or as another service contract. Note that, the methodology treats functional client queries as service contracts with simple “dummy” workflows, consisting of one task only, which is linked to the input and output conditions of the workflow.

Clients can employ the methodology for the generation of (composite and/or adapted) services that:

(1) Can interact successfully with the client request, or that

(2) Provide the same functional description (in terms of requested inputs and provided outputs) as the client query.

In the case of black-box requests, clients can enforce the first behaviour of the methodology (viz., (1)) by setting the inputs and outputs (IOs for short) of the dummy query task to the IOs of the query. Dually, the second behaviour of the methodology (viz., (2)) can be obtained by “flipping” the query IOs into the dummy query task.

If clients define queries as service contracts, (1) can be obtained by feeding the contract as is to the methodology. Furthermore, (2) can be obtained by replacing the behaviour information in the client contract with a dummy workflow, made of one task only. Similarly to the previous case, the IOs of this dummy task correspond to the OIs of the client service. Note that, this latter case outputs a service that requests at most the inputs provided by the query, and that generates at least the outputs requested by the query. However, it is likely that the output service does not “behave” like the client one.

It is important to note that, since the aggregation and adaptation processes use contracts to represent services, their integration is straightforward. On the one hand, service aggregation may require some adaptation. In this case, the adaptation can be plugged with the aggregation for the customisation of services that do not
(fully) satisfy client requests. Furthermore, service adapters and/or adapted services can be located and then composed with other services. On the other hand, the (behavioural) adaptation process discussed in this thesis constructs the contract of the adapted service as the aggregation between the original service and the generated adapter. However, note that the two techniques can also be employed as standalone processes. For example, clients can use the core aggregation process to construct the composite contract of a set of contracts they provide as input. Furthermore, clients can use the core adaptation technique to generate the contract of an adapter for two given services whose interaction locks.

In the following we briefly review the core of the aggregation and adaptation methodology through a simple application scenario. Assume a service developer is in possession of a BPEL process $C$, and that she wishes to locate advertised BPEL processes $S$ that can successfully interant with $C$. In other words, the service $S$ should provide all the inputs needed for the execution of $C$ and vice versa, as well as their composition should be lock-free. Assume further that the developer provides the service $C$ as input of the methodology, and that the methodology uses a registry $R$ of service contracts for the location of services. Furthermore, we consider that all services are annotated with (OWL) ontology information.

Roughly, the methodology first constructs the contract of $C^1$ by transforming the BPEL process into a corresponding YAWL workflow through the BPEL2YAWL translator. Next, it generates the execution traces of $C$ through a reachability analysis of the workflow of $C$. Then, it matches them against the execution traces of the services in $R$ with the purpose of individuating sets of execution traces of the services in $R$ such that they collectively output all the inputs requested by $C$, and dually, the outputs of $C$ suffice for the execution of the respective traces. Each set of execution traces gives a candidate set of services.

Consider now one such candidate set of services. Following, the methodology aggregates the services in the candidate set with the service $C$. This is basically achieved in two steps. First, the control-flow of each service is used to generate the main control-flow dependencies of the aggregate. Second, the data-flow mapping linking workflow tasks of different services (that was previously obtained during the process of matching service traces) leads to adding further control-flow constraints between the tasks of the composite. The result of this phase is the contract (call it $SC$) of the composite service. We recall that, together with the generation of $SC$, the aggregation process also builds the contract $S$ corresponding to the aggregation of the services in the candidate set only.

Next, the methodology formally checks whether $SC$ locks through a reachability analysis of its workflow. Should $SC$ be lock-free, the methodology deploys the contracts of $SC$ and $S$ as BPEL processes. Roughly, the core of the deployment phase parses the YAWL workflows with respect to the patterns defined by the BPEL2YAWL. The two BPEL processes make the answer to the client request.

---

1For simplicity we call it $C$ as well.
Otherwise, the methodology attempts to generate the contract of an adapter $A$ that lets $S$ and $C$ to interact successfully.

In this case, the (behavioural) adaptation process builds from the service execution trees (viz., the tree representation of the service execution traces) of $S$ and $C$ the execution trees of their duals with respect to each other. Then, from these adapter-views of $S$ and $C$, the adaptation constructs the execution tree of an adapter $A$, and then the workflow of $A$. Finally, the methodology validates the interaction of $A$ with $S$ and $C$ through a reachability analysis of the workflow of their aggregation. Should the composite have at least one lock-free trace, the methodology replies with the BPEL processes obtained through the deployment of the contracts of $A$, and of the composition between $A$ and $S$.

In the following we briefly review the main features of our methodology:

- It defines a way to locate, aggregate, and adapt service contracts (e.g., translations of BPEL processes augmented with OWL ontology information) so as to satisfy both functional and behavioural client requests,
- It supports service location, aggregation, and adaptation at the level of service execution traces (and not simply at the entire service level),
- It employs ontology information to locate services and to generate data-flow mappings between them, as well as to overcome semantic mismatches among service parameters,
- It sets the basis for the formal verification of services (e.g., BPEL processes), and for the development of heterogeneous services, as it employs YAWL for describing the service behaviour, as well as
- It provides an automated pattern-based compositional translator of BPEL processes into YAWL workflows, which
  - Copes with all types of BPEL activities (including flows with synchronisation links, and scopes), and
  - Handles exceptional behaviour – events, faults and (explicit) compensation.

Further investigation is mainly needed for the semi-automated generation of service contracts from Web services, and dually, for the deployment of service contracts as Web services.

On the one hand, in the current methodology the human designer is in charge of augmenting contracts with ontology information provided no such information exists in the services being translated, and of defining sets of equivalent ontology concepts so as to cope with cross ontology mappings. Overcoming such limitations calls for future research e.g., investigating the semi-automated derivation of (OWL)
ontology-information from Web service descriptions. Furthermore, there is need for further translations between languages describing service behaviour (e.g., OWL-S, WSCDL, WSMO) and YAWL.

On the other hand, the aggregation methodology can be enhanced to deploy service composers that orchestrate the participant services into the composite service (as indicated in Section 3.10). Intuitively, a service composer can be constructed from the composite contract using the dual-views introduced in Chapter 4. Furthermore, the compositional pattern-based nature of BPEL2YAWL can be exploited for the definition of an inverse YAWL2BPEL translator.

Two possible extensions of the adaptation process consist in the development of techniques for i) the generation of all possible adapters that can successfully overcome behavioural mismatches due to different operation orderings in the interacting partner processes, as well as for ii) the generation of adapters that mediate several interacting parties. Another possible extension involves tackling signature mismatches (e.g., different operation names), as well as mismatches among the ontologies employed by the service contracts.

As another important direction for future work we look forward towards engineering and (implementing) the methodology and experimenting it, as well as its deployment (e.g., following the general guidelines illustrated in Section 3.8) as a single tool supporting the disciplined, semi-automated aggregation and adaptation of Web services.
Bibliography


[31] DAML-S Coalition. DAML-S 0.9 draft release. (http://www.daml.org/services/daml-s/0.9/).


[69] OMG. Component object request broker architecture (corba). (http://www.corba.org/).


[71] OWL-S Coalition. OWL-S: Semantic Markup for Web Services Version 1.1. (http://www.daml.org/services/owl-s/1.1/overview/).


[79] P. Rajasekaran, J. A. Miller, K. Verma, and A. P. Sheth. Enhancing Web Services Description and Discovery to Facilitate Composition. In J. Cardoso and


[91] E. Verbeek and W. M. P. van der Aalst. Woflan 2.0: A petri-net-based work- 
flow diagnosis tool. In Nielsen, M. and Simpson, D., editors, Lecture Notes in 
Computer Science: 21st International Conference on Application and Theory 
of Petri Nets (ICATPN 2000), Aarhus, Denmark, June 2000, volume 1825, 

[92] W3C. Web Service Choreography Interface (WSCI) 1.0. World Wide Web 


(http://www.w3.org/TR/2000/REC-xml-20001006).

[95] W3C. Simple Object Access Protocol (SOAP) 1.2, W3C working draft, 
17 December 2001. 2001. (http://www.w3.org/TR/2001/WD-soap12-part0- 
20011217/).

[96] F.-Y. Wang, Y. Gao, and M. Zhou. A Modified Reachability Tree Approach 
to Analysis of Unbounded Petri Nets. IEEE Transactions on Systems, Man 

Analysis of Web Services Composition Languages: The Case of BPEL4WS. In 
I.-Y. Song, S. W. Liddle, T. W. Ling, and P. Scheuermann, editors, Proceedings 
of the 22nd International Conference on Conceptual Modeling, volume 2813 of 

trees for Petri nets. In Regional Control Conference, Brooklyn, NY, July 24- 

[99] WSCDL Coalition. Web Services Choreography Description Language Version 
1.0. (http://www.w3.org/TR/ws-cdl-10/).

[100] WSDL Coalition. Web Service Description Language (WSDL) version 1.1. 
(http://www.w3.org/TR/wsd1).


[102] WSLA Coalition. Web Service Level Agreement (WSLA) Language Specifi- 

(http://www.wsmo.org/TR/d2/v1.2/).


