Università di Pisa

Dipartimento di Informatica

Technical Report: TR-06-12

Contract-based Service Aggregation

Antonio Brogi

Razvan Popescu

Computer Science Department, University of Pisa, Italy

July 12, 2006

Address: via F. Buonarroti 2, 56127 Pisa, Italy. Tel: +39 050 2212700 Fax: +39 050 2212726

Contract-based Service Aggregation

Antonio Brogi

Razvan Popescu

Computer Science Department, University of Pisa, Italy

July 12, 2006

Abstract

We present a methodology for the automated selection and aggregation of (Web) services with the purpose of satisfying client queries. A key ingredient of our approach is the notion of *service contract*, which consists of *signature* (WSDL), *ontology information* (OWL), and *behaviour specification* (YAWL).

The methodology inputs a registry of service contracts and a client service contract, and it automatically generates aggregated contracts that fulfil the request. By trace inspection we first individuate 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.

1 Introduction

Service aggregation is one of the main issues of the Service-oriented Computing (SoC) paradigm [27] and it deals with building new services from existing ones. Current 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. On the one hand, the industry (mainly) promotes BPEL [7] as a language to express compositions

of WSDL [40] services, yet, the designer is in charge of finding and aggregating the appropriate services. On the other hand, the semantic Web initiative argues for the use of ontology languages such as OWL-S [25] in order to enhance and to automate the aggregation process. Most automation-oriented approaches employ A.I. techniques such as planning (e.g., [6, 22, 32, 42]), 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 written with different service description languages.

Our long-term objective is to develop a general methodology for deploying (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 this paper we present a (Web) service aggregation methodology that, given a registry of (advertised) service contracts and a client service contract, automatically generates compositions of contracts that satisfy the client request. Service contracts include a signature (expressed as a WSDL interface), ontology information (described with OWL, for example), as well as a description of the service behaviour (expressed by a YAWL [34] workflow). Note that we use the term *contract* to denote a "rich service description" (e.g., as in [24]) and not "an agreement among multiple parties" (e.g., SLA).

The methodology we propose tackles the aggregation at the *execution trace* level and not at the entire service level. Informally, an execution trace consists of a sequence of "atomic work units" executed during a service execution instance. We initially individuate candidate sets of services that may be aggregated in order to satisfy the client service. This phase matches the execution traces of the services in the registry with the execution traces of the client service. Note that the methodology also supports the matching of a subset of client traces only. Assume a client service having two execution traces, one for reserving flight tickets and another one for booking hotel rooms. If we assume further that the registry contains one service only, which offers

flight tickets, then only one client trace can be matched and fulfilled. A candidate set is characterised by the fact that its traces together with the client's matched traces form a "closed workflow" in the sense that the inputs set needed by all traces collectively is contained in the outputs set generated by all such traces. Next, for each candidate set we generate the contract of the composed service by firstly performing a control-flow and then an (ontology-aware) data-flow analysis of the behaviour of the contracts to be aggregated. 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. Finally, for each aggregated contract we generate its execution traces in order to verify its lock-freedom and whether the matched client traces can be actually fulfilled. Informally, we consider a client trace to be satisfied if the aggregated service has at least one successful execution trace such that all atomic units of work executed in the chosen client trace are executed by the corresponding trace of the aggregate as well. The approach we propose here tries to satisfy the maximum number of client traces. If all client traces are fulfilled we say that the aggregation *fully satisfies* the client request. Otherwise we say that the aggregation *partially satisfies* (if only some client traces are fulfilled), or that it *does not satisfy* (if none of the client's traces is fulfilled) the client request.

To the best of our knowledge our methodology 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 at the entire service level),
- it relies on service contracts and execution traces, which can be both generated off-line,
- 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.

Although this paper focuses on the methodological aspects of our approach, a section of the paper is devoted to discuss the main middleware aspects regarding the deployment of the aggregation phases as well as of the entire aggregation process as Web services.

The rest of the paper is organised as follows. In Section 2 we introduce YAWL. In Section 3 we present a motivating example that will be used throughout the paper to illustrate the methodology. Section 4 gives an overview of the aggregation methodology, while the service contracts are introduced in Section 5. Section 6 is dedicated to describing the aggregation methodology. In Section 7 we discuss the main middleware aspects of the methodology. Finally, in Section 8 we briefly review related work, and in Section 9 we present some concluding remarks.

2 Background: Yet Another Workflow Language (YAWL)

In this Section we briefly describe YAWL [34] starting with an informal description of a couple of workflows, followed by some insights on the key elements and features of the language.

Figure 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 [34]. Hence, 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*. Note that in addition to the YAWL representation of a workflow, we graphically represent the inputs and the outputs of the tasks in the workflow. 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.

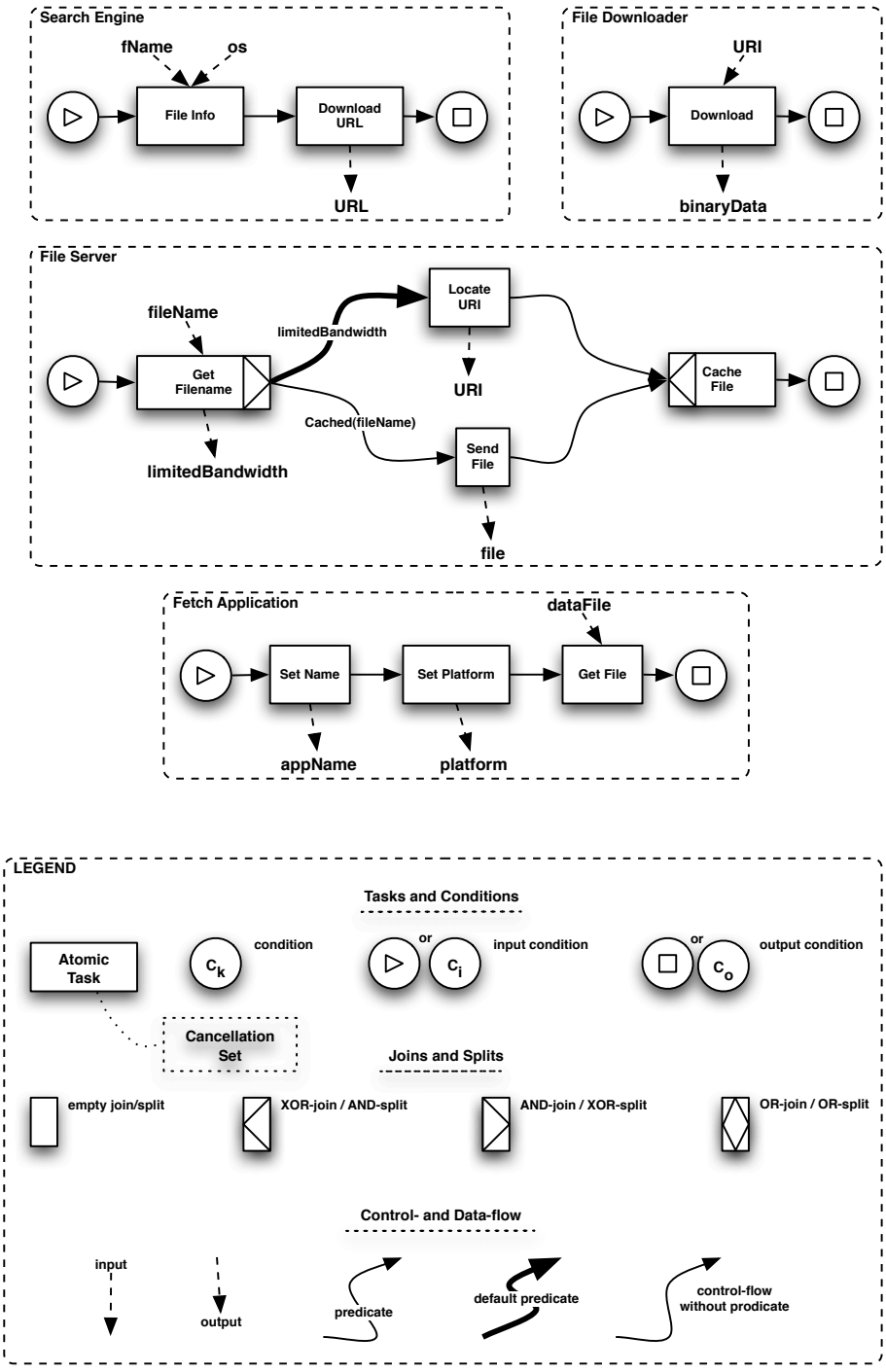

Figure 1: Examples of YAWL workflows.

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.

We consider that YAWL is a promising candidate to be used as an abstract workflow language for describing service behaviour. 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 [35]. 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. With respect to process algebras, YAWL features an intuitive (graphical) representation of services through workflow patterns. Furthermore, as illustrated in [33], 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 vs. π-calculus may be found in [33]. 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.

From a control-flow perspective, a YAWL file describes a *workflow specification* that consists of one or more *extended workflow net*s (or EWF-nets for short) arranged in a tree-like structure. An EWF-net is a graph where nodes are *task*s or *condition*s, and arrows define the control-flow relation. Each EWF-net has a single *input condition* and

a single *output condition*. A task Q is to be executed after another task P if there is an arrow from P to Q. 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. 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). 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 splits, 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. Finally, according to [34], if the task has an OR-join, then it 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.

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.

3 Motivating Example

Consider a client service, *Fetch Application*, whose YAWL behaviour is given in Figure 1, and suppose that the client wishes to use this service to download applications. Informally, *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 behaviours are given by the YAWL workflows presented in the top part of Figure 1. *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.

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 aggregation issues.

For simplicity, we shall consider *exact* matches [26] among parameters 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 inputs and outputs (IOs for short)

[26] 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 an *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, as such approaches view both the client request and the advertised services as black-boxes (i.e., behaviourless), 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* needed by the latter.

Many approaches to composition-oriented discovery of services [4, 5, 8, 16, 17, 19] 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 methodology and the client request might lock once again.

The aggregation methodology we describe in this paper looks for compositions of

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.

4 Overview of the Aggregation Methodology

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

0. **Service Translation**. This preliminary phase deals with translating real-world descriptions (e.g., BPEL + semantics, or OWL-S, etc.) 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 ontological information. A thorough analysis of how to transform BPEL specifications into workflow patterns can be found in [37]. 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 [26], 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 automat-

ically 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. Together 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 successfully) 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 our methodology 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 service can be deployed as a real-world Web service (i.e., described using OWL-S, or

BPEL + ontological information, etc.). 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.

5 Service Contracts

Currently, providers publish (purely syntactic) WSDL [40] advertisements to UDDI [12] 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 *semantic 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 descriptions. WSDL-S [3], OWL-S [25], SWSO [30], WSMO [41], or METEOR-S [29] annotate services with semantic information. BPEL [7], WSCDL [39], METEOR-S [2], OWL-S [25], SWSO [30], or recently YAWL [34] add protocol information to 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.

Our long-term goal is to build an aggregation methodology capable of composing services described using possibly different process/workflow modelling languages (e.g., BPEL [7], OWL-S [25], etc.), as well as of supporting multiple deployments of the aggregate as real-world services. The difficulties of achieving this aim mainly arise from the fact that most of the existing service description languages lack *ontological*

information and/or *formal semantics*.

As a consequence, in order to tackle these two issues we consider services that are described by *contracts* [24], and we argue that contracts should in general include different types of information: (a) *Signature*, (b) *Ontology information*, (c) *Behaviour*, and (d) *Extra-functional properties*.

The signature can be expressed in terms of WSDL, which is the current standard for describing Web service interfaces. Following [25], we argue that (WSDL) signatures should be enriched with ontological information (e.g., expressed with OWL [21] or WSDL-S [3]) to better capture the semantics of services, and necessary to automatise the process of overcoming signature mismatches, as well as service selection and composition. Still, the information provided by the signature and ontological description levels is necessary but *not* sufficient to ensure a correct inter-operation of services.

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 [24], 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 Section 2, we consider that YAWL is a promising candidate to be used as an abstract workflow language for describing the service behaviour.

Finally, we argue that service contracts should expose, besides annotated signatures and behaviour, also so-called extra-functional properties, such as performance, reliabil-

ity, or security. (We will not however consider these properties in this work, and leave their inclusion into the aggregation methodology as future work.)

6 Description of the Aggregation Methodology

We start with the description of the reachability analysis (Subsection 6.1) and of the service execution traces (Subsection 6.2), 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 6.3), the core aggregation and the generation of the aggregated contract (Subsection 6.4) as well the contract validation phase (Subsection 6.5). Subsection 6.6 analyses the complexity of our approach.

6.1 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:

- *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.

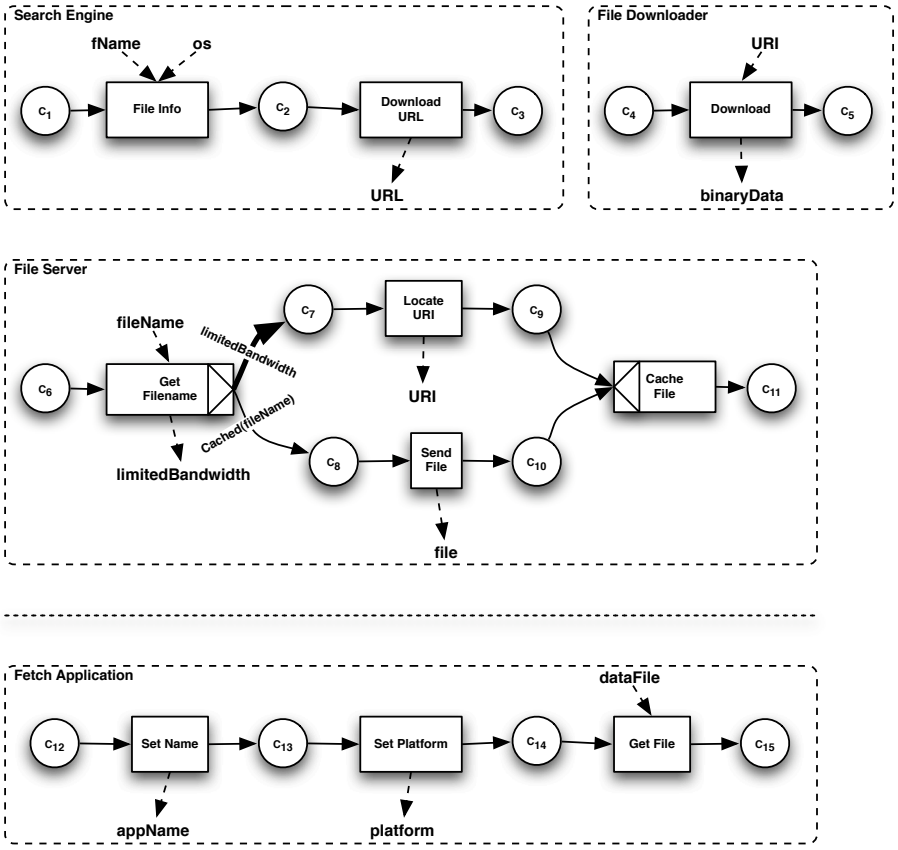

Figure 2: Motivating example workflows with explicit conditions.

- *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 2 obtained from the workflows in Figure 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 [43] 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

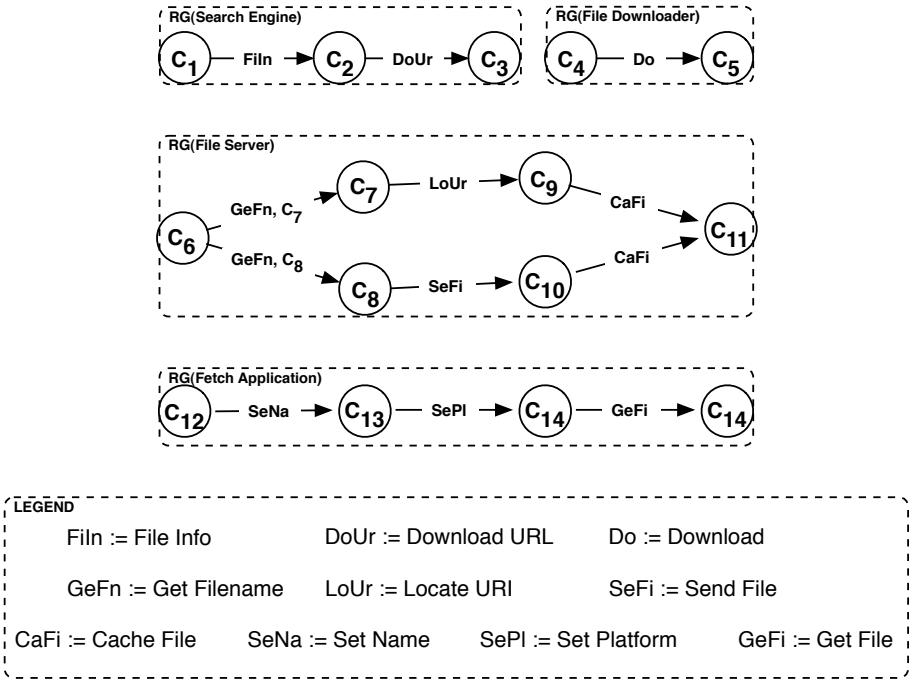

Figure 3: RGs of the four workflows in Figure 2.

as $C_i + ... + C_j$. 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 depicts the RGs corresponding to the four workflows in Figure 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 [43].

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

- **initial marking** M_i: the marking without incoming links. It contains only the initial condition of the workflow.

- **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 [34].
- **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 [15] 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 ω-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 ω-symbol is called ω-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 [14] 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 ω-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 ω-symbol. In order to tackle these properties, Wang et al. [36] formalised the *modified reachability tree* (MRT), which uses ω-numbers instead of ω-symbols. Similarly to FRTs, a MRT ω-marking contains at least one ω-number. ω-numbers denoted by $k\omega_n + q$ are subsets of integers of the form $\{ik + q \mid i \geq n\}$, where the *base* $k \in N^+$, and the *least bound* and respectively, the *reminder* $n, q \in Z$ and they can capture more information on the

structure of the infiniteness than ω-symbols. For example, a place in a marking to which it corresponds a $2\omega_1$ ω-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.

In this paper we propose the usage of the MRT algorithm defined in [36][1] to build the MRT of a YAWL workflow with the purpose of analysing YAWL workflows, and consequently for the analysis of service behaviours. However, due to space limitations, and 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 is the base of the RG, offer the same kind of information due to the fact that they both contain the same markings. All the example employed in this paper have bounded representations. Our main concern in this paper 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.

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

6.2 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 $\langle$*Preconditions, Needed Inputs, 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

[1]Slightly adapted so as to cope with YAWL workflow nets instead of Petri nets.

Search Engine $\{T_{SE}\}$**:** $<\{C_1, C_2, C_3\}$, $\{$*fName, os*$\}$, $\{$*URL*$\}>$.
File Downloader $\{T_{FD}\}$**:** $<\{C_4, C_5\}$, $\{$*URI*$\}$, $\{$*binaryData*$\}>$.
File Server $\{T^1_{FS}, T^2_{FS}\}$**:** $<\{C_6, C_7, C_9, C_{11}\}$, $\{$*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 1: TTs of the example workflows.

Workflow with cycle $\{T_1, T_2\}$**:** $<\{C_i, C_2, C_o\}$, $\{a\}$, $\emptyset>$, $<\{C_i, C_1, C_2, C_3, C_o\}$, $\{a\}$, $\emptyset>$.

Table 2: TT for the workflow in Figure 4.

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 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 4 we consider only two successful traces, as given by the TT in Table 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 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".

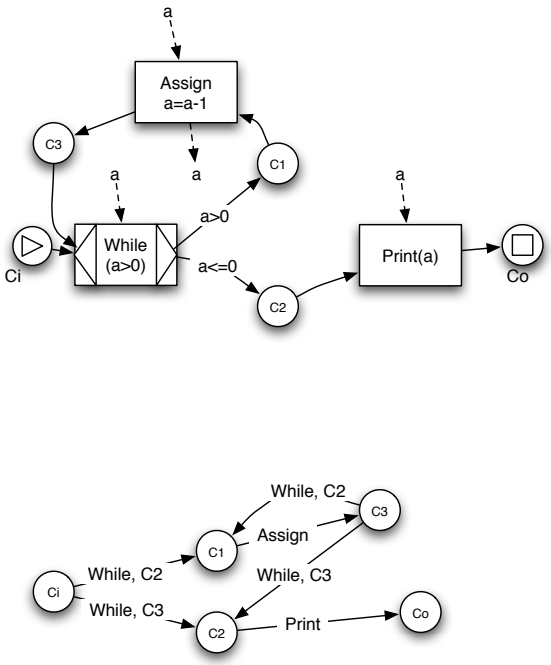

Figure 4: Workflow with cycle.

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 *"limitedBandwidth OR (NOT Cached(fileName))"*. To do so, by exploiting the usability of the YAWL predicates to enable tasks [34], 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 "true" condition. Furthermore, output places of tasks having an EMPTY- or an AND-split get an always "true" condition (e.g., C_9). In the case of a XOR-split, we consider an output condition to be true provided "*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 "*limitedBandwidth OR ((limitedBandwidth = default) AND (NOT Cached(fileName)))*", or simply "*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 "*(Cached(fileName) AND (NOT limitedBandwidth)) OR ((Cached(fileName) = default) AND (NOT limitedBandwidth))*", or simply "*(Cached(fileName) AND (NOT limitedBandwidth))*". Last but not least, for a task having an OR-split, we consider an output condition to be true if and only if "its corresponding predicate is true, *or* the respective predicate is the default one and all other predicates of the considered tasks are false" [34].

6.3 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. 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*. 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 [25], 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 (*exact match*), or O_i represents a sub-concept of I_j ("O_i plugs-in I_j", or equivalently "I_j subsumes O_i"). 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 5. It shows that there are three candidate sets for fully satisfying the client request: (a) $\{T_{FD}, T_{SE}\}$, which corresponds to executing the *File Downloader* and the *Search Engine* services, (b)

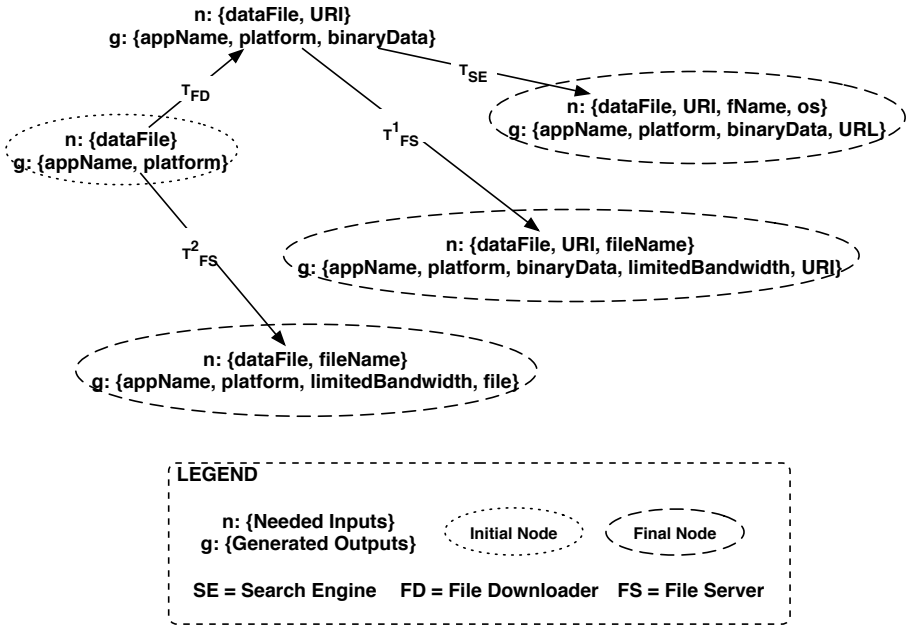

Figure 5: *Matchmaker Graph* for our example.

$\{T_{FD}, T^1_{FS}\}$, which corresponds to executing the *File Downloader* and the *File Server* services, and (c) $\{T^2_{FS}\}$, 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 (*exact match*), or O_i represents a sub-concept of I_j ("O_i plugs-in I_j", or equivalently "I_j subsumes O_i") [25]. 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 $pType_i$

Candidate Set ∪ 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: 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 4: Data-flow mapping for {*File Server, File Downloader, Fetch Application*}.

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* 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, 4, and 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 aggre-

{File Server, Fetch Application}:
Get Filename(fileName) <> Set Name(appName);
Get File(dataFile) <> Download(binaryData).

Table 5: Data-flow mapping for {*File Server, Fetch Application*}.

gation 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 $\{T_{FD},\ T^1_{FS}\}$ we have to aggregate the contract of the client *Fetch Application* service with the the contracts of the *File Downloader* and *File Server* services.

6.4 Core Aggregation and Contract Generation

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. 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). The second step translates the initial flow dependencies of each workflow in terms of the newly added IC and OC dummies. The third step relates IDs and ODs of tasks belonging to (possibly) different workflows by taking into account the data-flow mapping. The fourth and final step clears the aggregated contract of redundant dummies and control constructs. The four steps are detailed hereafter.

6.4.1 Task Expansion

The Task Expansion starts by considering the 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,

(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 6 describes the process 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.

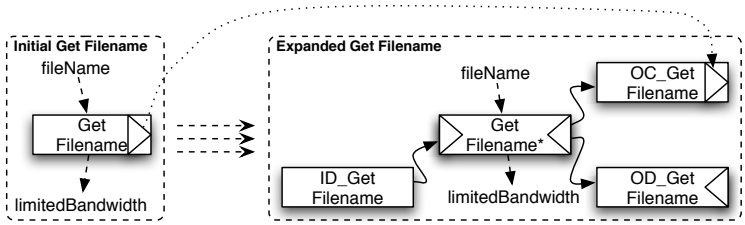

Figure 6: Expansion of the *Get Filename* task of the the *File Server* service.

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 ICs of all the workflows to be aggregated. Dually, OC_A has an AND join in order to wait for the OCs 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.

6.4.2 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. 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 7.

For example, the initial control-flow link between *Get Filename* and *Locate URI* has been translated into a link between *OC_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*

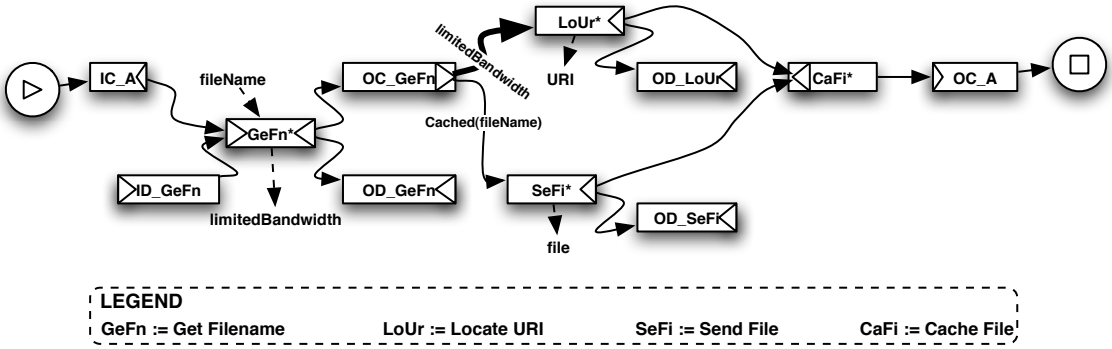

Figure 7: Control-flow analysis for the *File Server* service.

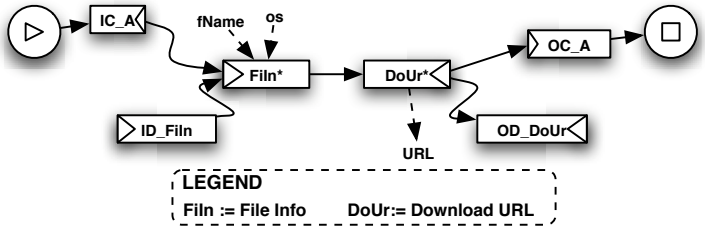

Figure 8: Control-flow analysis for the *Search Engine* service.

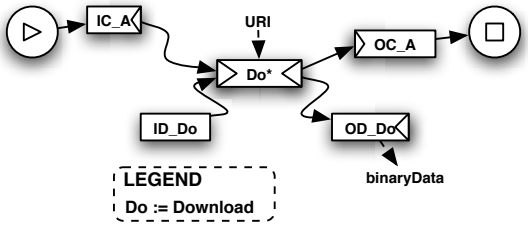

Figure 9: Control-flow analysis for the *File Downloader* service.

for execution. Dually, the execution of *Cache File* is interpreted as the termination of the *File Server* service.

The control-flow analysis for the *Search Engine*, *File Downloader*, and *Fetch Application* services are shown in Figure 8, 9, and 10, respectively.

6.4.3 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.

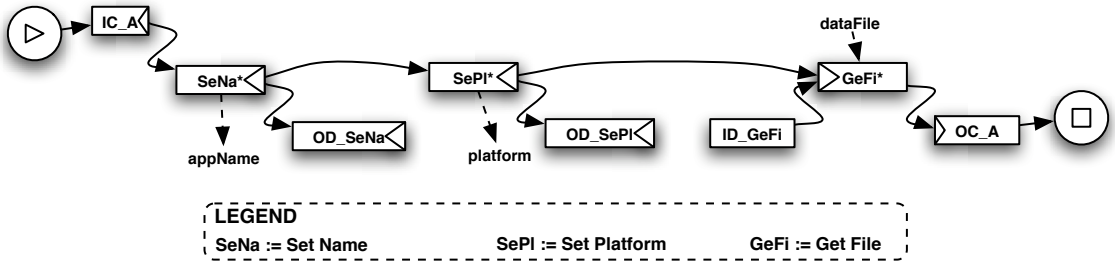

Figure 10: Control-flow analysis for the *Fetch Application* service.

Consider the data-flow mapping 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. Then, 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 unique.

2. For each pair $((W, T, i), (Z, S, o))$ in M, add to A a link from OD_S to T_i.

Figure 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 in Figure 12, 13, and 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.

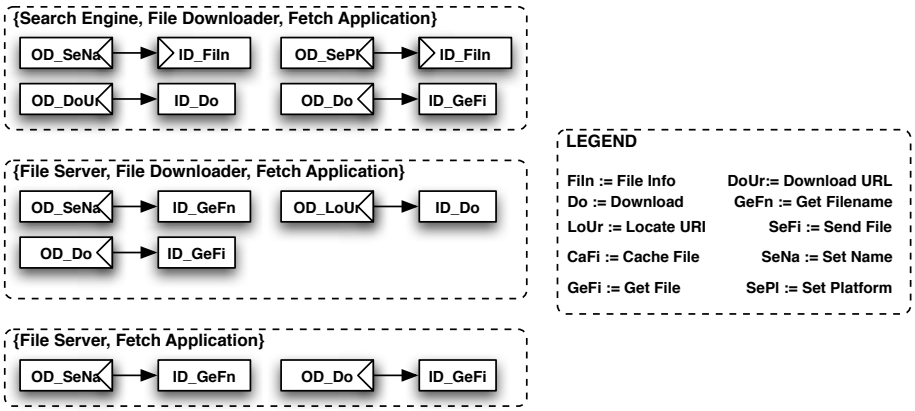

Figure 11: Data-flow analysis for the three possible scenarios of our example.

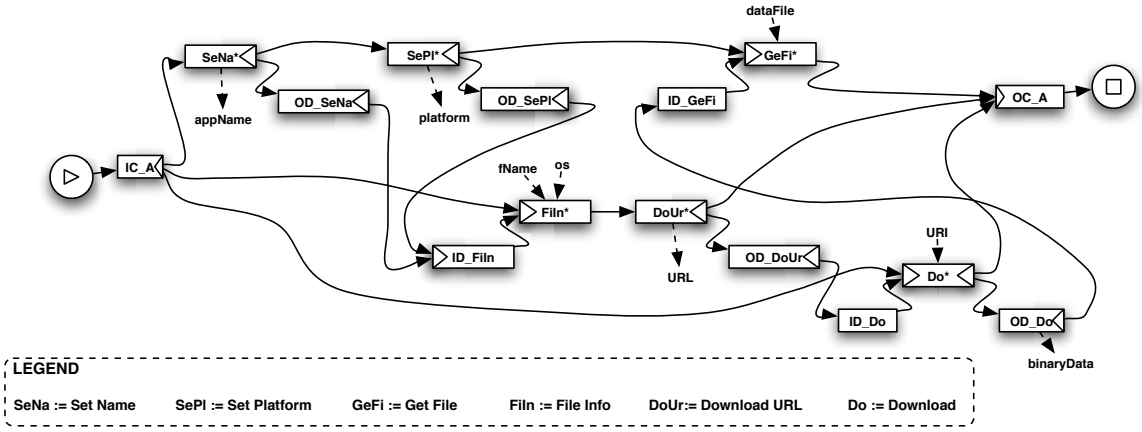

Figure 12: $AggS_1$: Workflow obtained by aggregating the *Search Engine*, the *File Downloader*, and the *Fetch Application* workflows.

6.4.4 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. Please note that the optimisation is not concerned with generating the "optimal aggregated workflow". 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 $< join_{iD}, join_T >$ matches of the following – $\{< EMPTY, EMPTY >, < EMPTY, \alpha >$

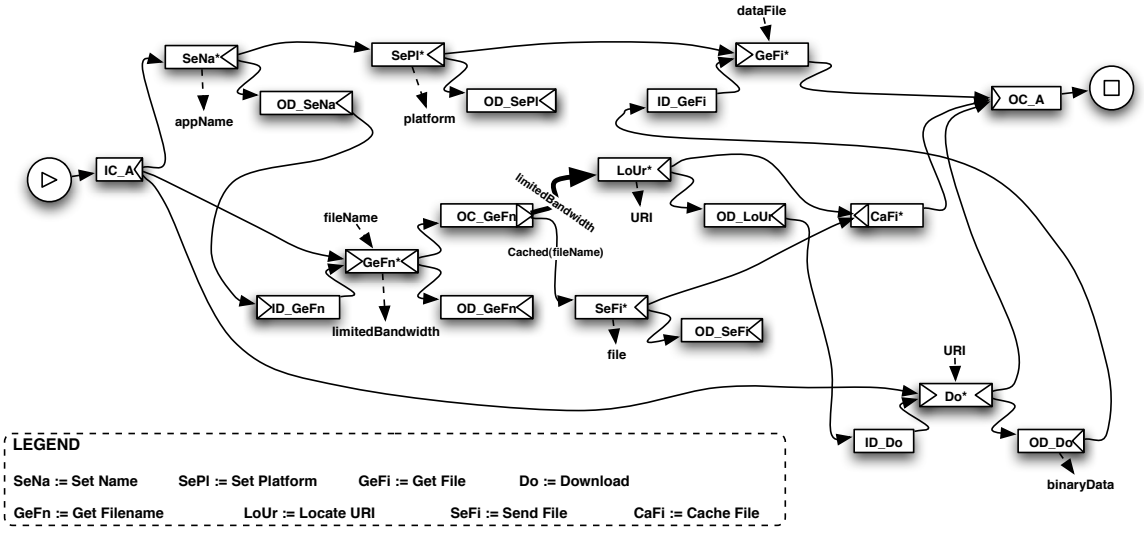

Figure 13: $AggS_2$: Workflow obtained by aggregating the *File Server*, the *File Downloader*, and the *Fetch Application* workflows.

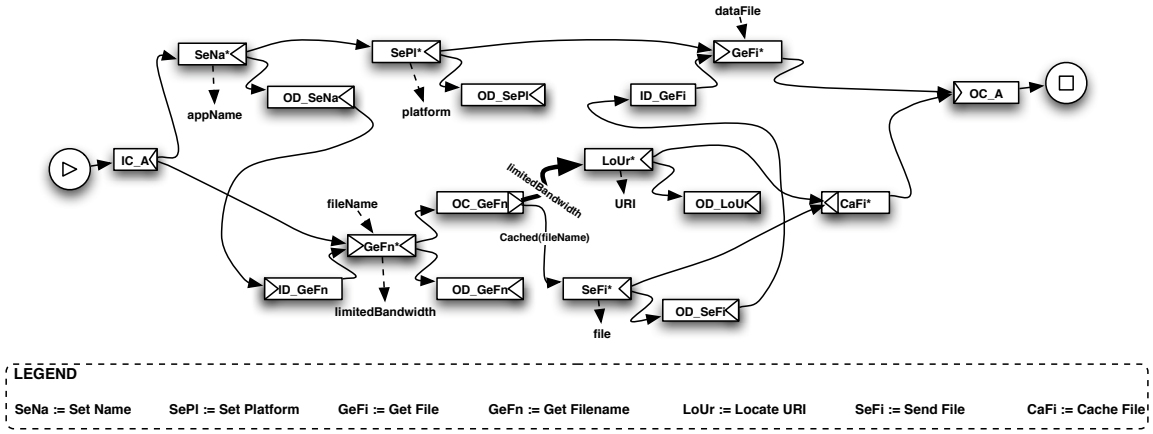

Figure 14: $AggS_3$: Workflow obtained by aggregating the *File Server* and the *Fetch Application* workflows.

$, < \alpha, \alpha >\}$ –, where $\alpha \in \{AND, XOR, OR\}$. 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 $< join_{iD}, join_T >$ matches $< \alpha, EMPTY >$, then iD is absorbed into T with the observation that T inherits the join of iD (i.e., $join_T := join_{iD}$). The scenario is dual for absorbing output dummies. This criteria can be applied for clearing all IDs and ODs of the aggregated services depicted in Figures 12–14.

Join/Split elimination. A $join_T \neq EMPTY$ has to be set to EMPTY provided T has *only*

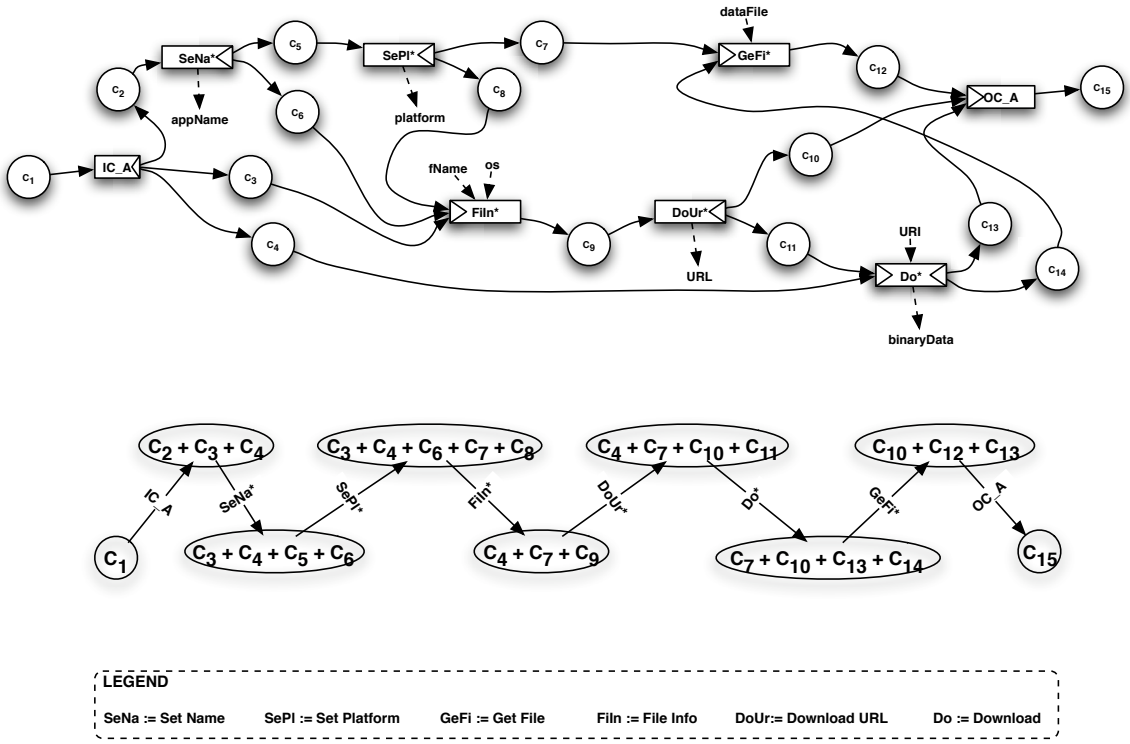

Figure 15: Optimised workflow and RG for the $AggS_1$ service.

one incoming link. The dual (i.e., the "reset" of $split_T$ given T has *at most one* outgoing link) is resolved in a similar way. For example, the $SetPlatform^*$, $GetFilename^*$, and $SendFile^*$ tasks of the aggregated service in Figure 13 get their AND-splits reset to EMPTY ones after previously absorbing their ODs.

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 15, 16, and 17, respectively.

6.5 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

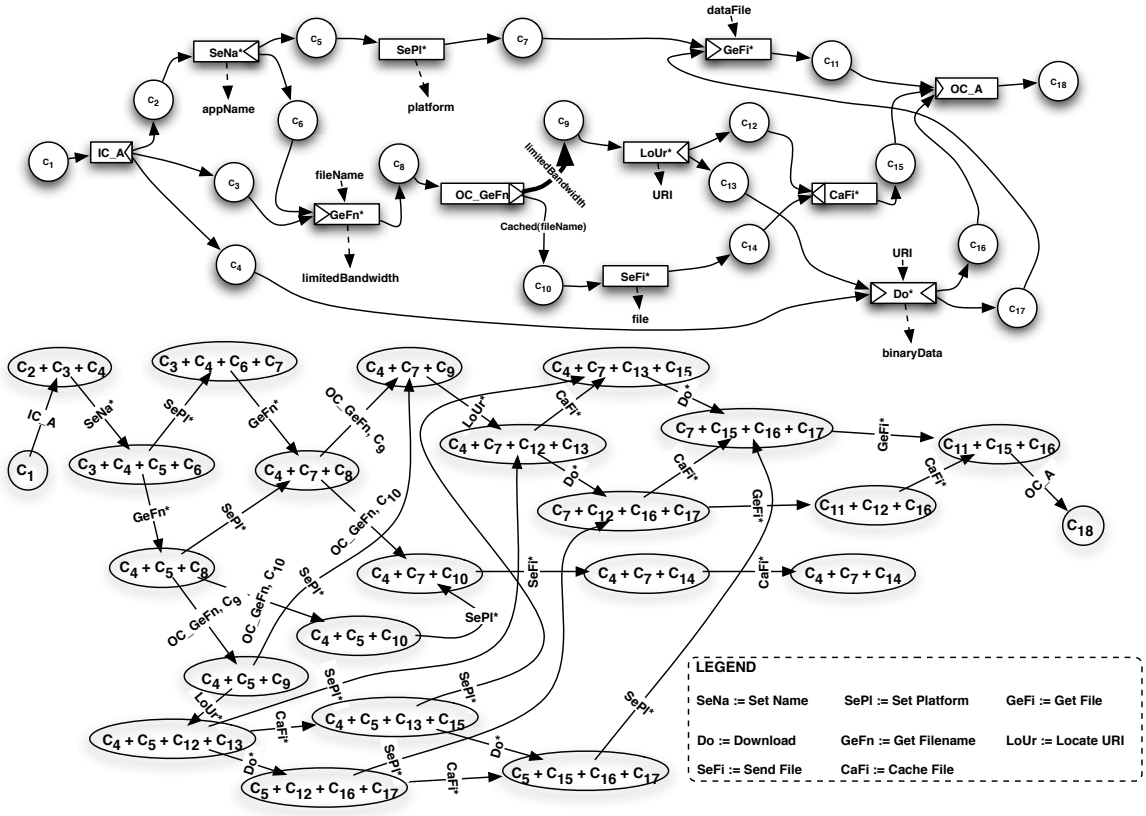

Figure 16: Optimised workflow and RG for the $AggS_2$ service.

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: (1) `YES: the aggregation of the services in A fulfils`

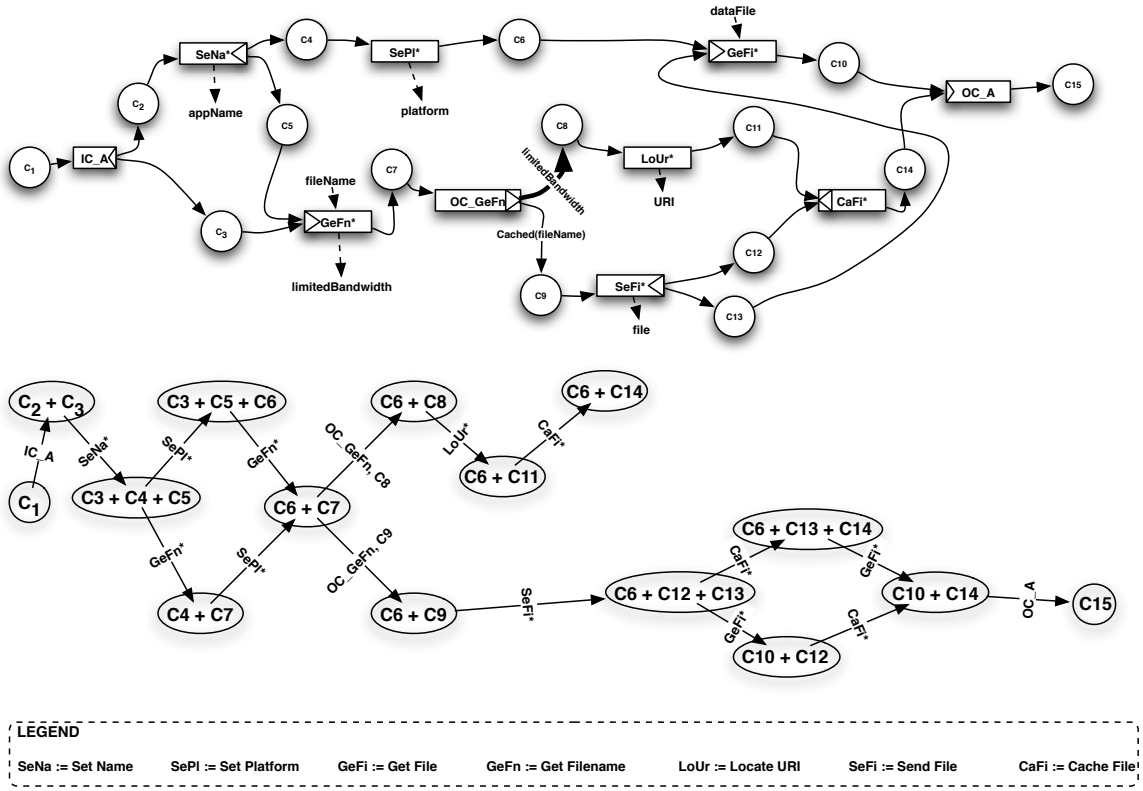

Figure 17: Optimised workflow and RG for the $AggS_3$ service.

`the trace u of the C client service.` – if `u` is not constrained by any preconditions set, (2) `MAYBE: the aggregation of the services in A may fulfil the requested trace u of the C client service [with condition Precond_1` $\wedge \ldots \wedge$ `Precond_N]` – if `u` is conditioned by at least one precondition set `PS_k`, and the logical expression `Precond_1` $\wedge \ldots \wedge$ `Precond_N` is obtained by the conjunction of the conditions in `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 our methodology 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

$AggS_1$: <{C_1, C_2, …, C_{15}}, {*fName, os, dataFile, URI*}, {*appName, platform, URL, binaryData*}>.

Table 6: TT of the aggregated service $AggS_1$ (Figure 15).

$AggS_2$: <{C_1, C_2, …, C_9, C_{11}, C_{12}, C_{13}, C_{15}, C_{16}, C_{17}, C_{18}}, {*fileName, dataFile, URI*}, {*appName, platform, limitedBandwidth, URI, binaryData*}>.

Table 7: TT of the aggregated service $AggS_2$ (Figure 16).

$AggS_3$: <{C_1, C_2, …, C_7, C_9, C_{10}, C_{12}, C_{13}, C_{14}, C_{15}}, {*fileName, dataFile*}, {*appName, platform, limitedBandwidth, file*}>.

Table 8: TT of the aggregated service $AggS_3$ (Figure 17).

aggregation, and 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 6–8 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 1) we get that the set of tasks executed for satisfying the client request is $Tasks\_C$ = {*Set Name, Set Platform, Get File*}. Similarly, by inspecting the successful execution traces of the three aggregated services (note the three TTs Table 6, Table 7, and respectively Table 8), we obtain the following sets of tasks that have to be executed for the successful execution of the three aggregated services: $Tasks\_AggS_1$ = {*IC_A, Set Name*, Set Platform*, Get File*, File Info*, Download URL*, Download*, OC_A*}, $Tasks\_AggS_2$ = {*IC_A, Set Name*, Set Platform*, Get File*, Get Filename*, OC_Get Filename, Locate URI*, Cache File*, Download*, OC_A*}, and $Tasks\_AggS_3$ = {*IC_A, Set Name*, Set Platform*, Get File*, Get Filename*, OC_Get Filename, Send File*, Cache File*, OC_A*}. One should note that all three candidate aggregates *fully* satisfy the client request as the set $Tasks\_C$ is included in $Tasks\_AggS_1$, $Tasks\_AggS_2$, and $Tasks\_AggS_3$, respectively. Moreover, for the only successful trace (call it `T_FA`) of the *Fetch Application* client service, our aggregation methodology 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 `"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 $AggS_2$ (note Table 7). We do not output it as it refers the variable `limitedBandwidth` as well as the `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 `"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 $AggS_3$ (note Table 8). Similarly to the previous case, we do not output this condition.

6.6 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 6.1 and 6.2, 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 has the same order of complexity [36] of the algorithm for generating FRTs, unfortunately the *reachability problem* (also called *coverability problem*) for Petri Nets is known to be EXPSPACE-hard [13]. As described in Subsection 6.2, 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 6.3, 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 searched and so on, till 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 $O(\sum_{x=0}^{c-1}(C(c,x)\cdot O(2^N))$, that is, $O(2^{cN})$ 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 6.4, 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 $S{\cdot}T$ tasks, hence taking $O((S{\cdot}T)^2)$ in the worst case. Finally, the Contract Optimisa-

tion 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 $O((S{\cdot}T)^2)$.

- **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 argue 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.

7 Middleware Aspects

We recall that our long-term goal is to deploy the aggregation methodology described in this paper as a middleware that can be used by service developers (clients) to successfully aggregate services written using different service description languages. Our aim is to offer a platform for the flawless inter/intra enterprise application integration that is able to overcome interaction mismatches. Although this paper focuses on describing the aggregation methodology, we shall summarise hereafter the main middleware aspects of our approach.

The high-level view of the architecture we propose for the aggregation process can be seen in Figure 18. We plan to deploy each phase of the aggregation 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 our Java proof-of-concept prototype implementation of the CACG (e.g., the aggregation of the services in [11]) show that this phase of the aggregation process can indeed be

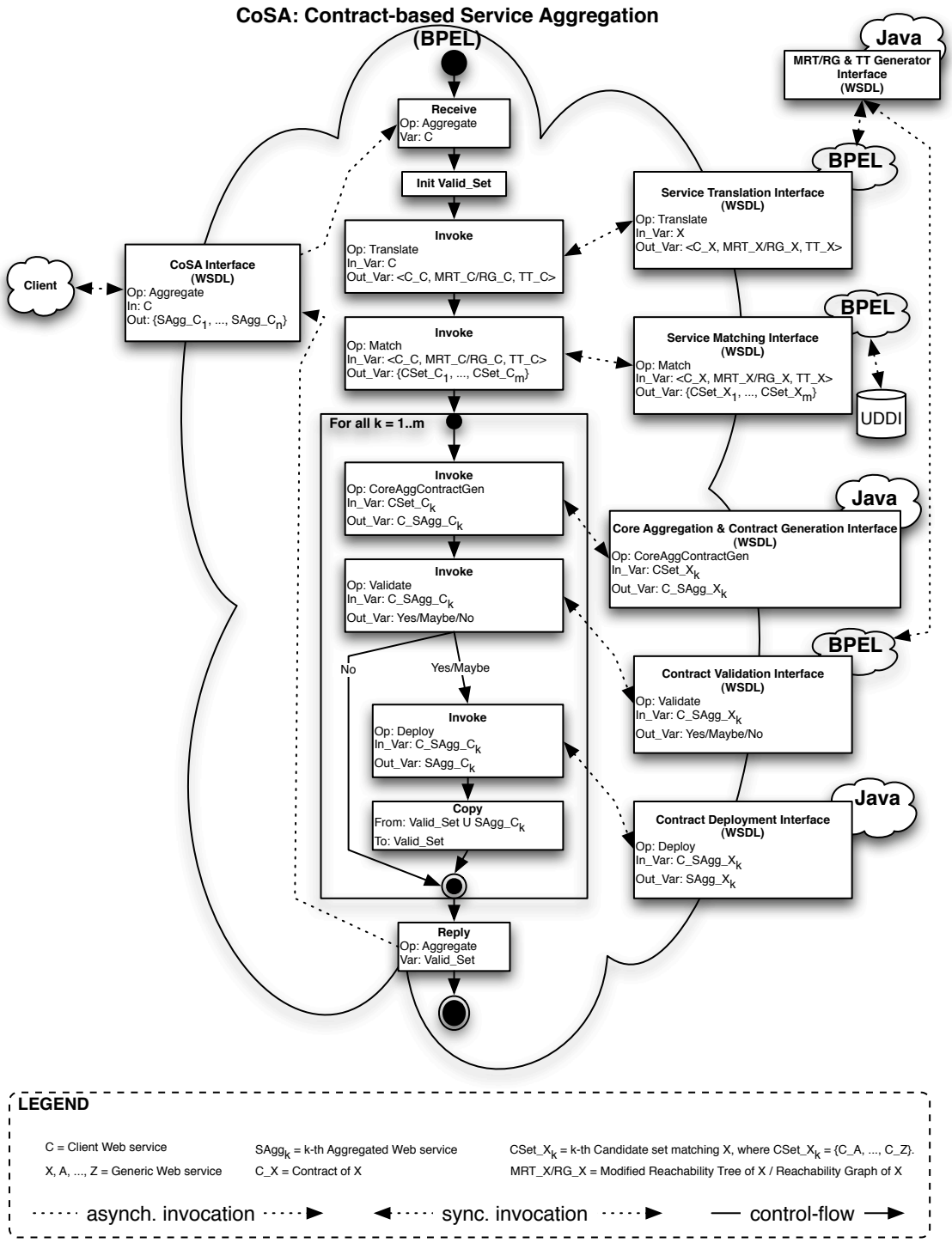

Figure 18: Deploying the aggregation methodology as a BPEL process.

automated. Another example is the program of Wong and Zhou [38] for the automated generation of MRTs. 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 by both the *Service Translation* and 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 methodology 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. Note that a BPEL process is deployed as a Web service as well, hence 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[2]). Figure 18 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 methodology described in this paper. Note that each aggregation 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 a 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. A first concrete example of automated translation of BPEL processes into YAWL workflows is described in [9]. A distinguishing feature of the BPEL2YAWL translator of [9] is that it handles all types of

[2]http://www.activebpel.com

BPEL activities, as well as events, faults and (explicit) compensation. Moreover, while YAWL does not model explicitly error/unexpected behaviours, our BPEL2YAWL translator provides a YAWL pattern template for the BPEL scope, which further defines a Fault Handler to catch faults possibly raised by a process. 2) Each service implementation can be updated independently of the rest. 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., [16]) 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. Space limitations do not allow us to go further into any details, yet note that contracts as well as traces of the advertised services can be generated off-line and stored into a "contract registry", 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 please 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 may either manually deploy the composed BPEL

process (e.g., in the ActiveBPEL engine), or use a semi-automated tool such as the Oracle BPEL Process Manager[3].

8 Related Work

In this section we briefly discuss other manual/semiautomatic/automatic approaches to Web service aggregation. At the end of the discussion we try to synthesise the (comparative) advantages of our approach.

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 (BPEL4WS, or BPEL for short) [7]. 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. [44] define, for instance, 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.

Semiautomatic 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. [19]. 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. [23] model the control-flow of the desired composed service while service advertisements are described through

[3]http://www.oracle.com/technology/bpel/

their IOs only. The composition is done by matching requested operations with the advertised ones based on IOs and non-functional properties.

The **automatic** 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. [31] 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 [28] 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 A.I. approaches 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. [22] 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 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 [42] 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. [32] use non-deterministic transition systems to model both services 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. [6] model service and client behaviour as finite state transition systems in which a transition abstracts the IO messages and operations. The output is automatically generated by delegating the requested actions to ones of the advertised services. However, a downside of planning (besides computational cost) is that representing the goal is difficult and error-prone.

Several reviews accurately describe current trends in Web services composition. In [18], 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 + BPEL4WS" (i.e., the industrial approach). In [1], Aalst et al. present a comparison of BPEL, XLANG, WSFL, BPML and WSCI. They show the trade-off between block-structured languages (e.g., XLANG, BPML, and WSCI) and graph-based languages (e.g., WSFL is graph-based). An interesting comparison between BPEL and DAML-S is provided by [20], while another one between BPEL and WSCI is given in [45]. 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 [37].

A preliminary version of the aggregation methodology described in this paper has been presented in [11]. The present paper substantially extends [11] by adding a matchmaker to select the set of services to be aggregated, and by introducing a validity check of the aggregated service. 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 automatic 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, and (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.

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 6.6) 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.

9 Conclusions

In this paper we described a methodology for aggregating services with the goal of satisfying a client request expressed as another service. The long-term goal of our methodology is to aggregate services written with different service description languages such as BPEL [7] or OWL-S [25]. A key ingredient of our framework is the notion of service contract consisting of a signature, an ontology description and 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 aggregating services written in different languages, and for multiple deployments of the aggregated service. A good candidate for a language to describe the ontology information is OWL, and parameter matching algorithms such as [26] 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 [34] 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.

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 [36] is a very useful tool that can be successfully employed for analysing service properties such as reachability or lock-freedom, and so on. 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 ω-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 `YES` or `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 *successful* (or that the client is *fully satisfied*) if all client traces are satisfied, *partially successful* (or that the client is *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 *failure*. The output of our

algorithm is a list of successful aggregations ordered by the number of unconstrained satisfied client traces (i.e., `YES` answers).

Future work will mainly be devoted to the semi-automatic derivation of service contracts from BPEL processes augmented with ontology information (using our prototype BPEL2YAWL translator) and to the implementation and deployment of the remaining aggregation phases as Web services. Another line of investigation is dedicated to extending the adaptation of signature and behavioural mismatches in contracts [10], and to applying it in this context.

References

[1] W. Aalst, M. Dumas, and A. Hofstede. Web service composition languages: Old wine in new bottles? In *Proceedings of Euromicro '03*, pages 298–307. IEEE Computer Society, 2003.

[2] R. Aggarwal, K. Verma, J. A. Miller, and W. Milnor. Constraint Driven Web Service Composition in METEOR-S. In *IEEE SCC*, pages 23–30. IEEE Computer Society, 2004.

[3] R. Akkiraju, J. Farrell, J. Miller, M. Nagarajan, M.-T. Schmidt, A. Sheth, and K. Verma. Web Service Semantics - WSDL-S Version 1.0. `http://lsdis.cs.uga.edu/library/download/WSDL-S-V1.html`.

[4] L. Aversano, G. Canfora, and A. Ciampi. An algorithm for web service discovery through their composition. In *IEEE International Conference on Web Services*, pages 332–, 2004.

[5] S. Bansal and J. Vidal. Matchmaking of Web Services Based on the DAML-S Service Model. In T. Sandholm and M. Yokoo, editors, *Second International Joint Conference on Autonomous Agents (AAMAS'03)*, pages 926–927. ACM Press, 2003.

[6] D. Berardi, G. D. Giacomo, M. Lenzerini, M. Mecella, and D. Calvanese. Synthesis of underspecified composite e-services based on automated reasoning. In *ICSOC '04: Proceedings of the 2nd international conference on Service oriented computing*, pages 105–114, New York, NY, USA, 2004. ACM Press.

[7] BPEL4WS Coalition. Business Process Execution Language for Web Services (BPEL4WS) Version 1.1. `ftp://www6.software.ibm.com/software/developer/library/ws-bpel.pdf`.

[8] A. Brogi, S. Corfini, and R. Popescu. Composition-oriented Service Discovery. In F. Gschwind, U. Assmann, and O. Nierstrasz, editors, *Proceedings of Software Composition '05, LNCS, vol. 3628*, pages 15–30, 2005.

[9] A. Brogi and R. Popescu. From BPEL Processes to YAWL Workflows. Technical Report, University of Pisa, April 2006. Available from: `http://www.di.unipi.it/~popescu/BPEL2YAWL.pdf`.

[10] A. Brogi and R. Popescu. Service Adaptation through Trace Inspection. In S. Gagnon, H. Ludwig, M. Pistore, and W. Sadiq, editors, *Proceedings of the First International Workshop on Service-Oriented Business Processes Integration (SOBPI '05), Amsterdam, The Netherlands*, pages 44–58, 2005. Available from http://elab.njit.edu/sobpi/sobpi05-proceedings.pdf.

[11] A. Brogi and R. Popescu. Towards Semi-automated Workflow-Based Aggregation of Web Services. In B. Benatallah, F. Casati, and P. Traverso, editors, *ICSOC*, volume 3826 of *LNCS*, pages 214–227. Springer, 2005.

[12] U. Coalition. The UDDI Technical White Paper. `http://www.uddi.org/`.

[13] J. Esparza and M. Nielsen. Decibility Issues for Petri Nets - a survey. *Journal of Informatik Processing and Cybernetics*, 30(3):143–160, 1994.

[14] A. Finkel. The Minimal Coverability Graph for Petri Nets. *LNCS*, 674:210–243, 1993.

[15] R. Karp and R. Miller. Parallel Program Schemata. *Journal of Computer and System Sciences*, 3:147–195, 1969.

[16] T. Kawamura, J. D. Blasio, T. Hasegawa, M. Paolucci, and K. Sycara. Public Deployment of Semantic Service Matchmaker with UDDI Business Registry. In S. A. McIlraith, D. Plexousakis, and F. van Harmelen, editors, *Proceedings of The Semantic Web ISWC'04, LNCS, Volume 3298*, pages 752–766, 2004.

[17] M. Kifer, R. Lara, A. Polleres, C. Zhao, U. Keller, H. Lausen, and D. Fensel. A Logical Framework for Web Service Discovery. In *ISWC 2004 Workshop on Semantic Web Services: Preparing to Meet the World of Business Applications*, volume 119, Hiroshima, Japan, 2004. CEUR Workshop Proceedings.

[18] J. Koehler and B. Srivastava. Web Service Composition: Current Solutions and Open Problems. In *ICAPS Workshop on Planning for Web Services*, pages 28–35, 2003.

[19] Q. Liang, L. N. Chakarapani, S. Y. W. Su, R. N. Chikkamagalur, and H. Lam. A Semi-Automatic Approach to Composite Web Services Discovery, Description and Invocation. *International Journal of Web Services Research*, 1(4):64–89, 2004.

[20] S. Liu, R. Khalaf, and F. Curbera. From daml-s processes to bpel4ws. In *RIDE*, pages 77–84. IEEE Computer Society, 2004.

[21] D. McGuiness and F. van Harmelen (Eds). OWL Web Ontology Language Overview. Web guide, February 2004. `http://www.w3.org/TR/owl-features`.

[22] S. A. McIlraith and T. C. Son. Adapting golog for composition of semantic web services. In D. Fensel, F. Giunchiglia, D. L. McGuinness, and M.-A. Williams, editors, *KR*, pages 482–496. Morgan Kaufmann, 2002.

[23] B. Medjahed, A. Bouguettaya, and A. K. Elmagarmid. Composing Web services on the Semantic Web. *The VLDB Journal*, 12(4):333–351, 2003.

[24] L. Meredith and S. Bjorg. Contracts and Types. *CACM*, 46(10), 2003.

[25] OWL-S Coalition. OWL-S: Semantic Markup for Web Services Version 1.1. `http://www.daml.org/services/owl-s/1.1/overview/`.

[26] M. Paolucci, T. Kawamura, T. Payne, and K. Sycara. Semantic Matchmaking of Web Services Capabilities. In I. Horrocks and J. Hendler, editors, *First International Semantic Web Conference on The Semantic Web, LNCS 2342*, pages 333–347. Springer-Verlag, 2002.

[27] M. P. Papazoglou and D. Georgakopoulos. Service-Oriented Computing. *Communication of the ACM*, 46(10):24–28, 2003.

[28] S. R. Ponnekanti and A. Fox. SWORD: A Developer Toolkit for Web Service Composition. In *The Eleventh World Wide Web Conference*, Honolulu, HI, USA, 2002.

[29] P. Rajasekaran, J. A. Miller, K. Verma, and A. P. Sheth. Enhancing Web Services Description and Discovery to Facilitate Composition. In J. Cardoso and A. P. Sheth, editors, *SWSWPC*, volume 3387 of *Lecture Notes in Computer Science*, pages 55–68. Springer, 2004.

[30] SWSO Coalition. Semantic Web Services Ontology (SWSO) Version 1.0. `http://www.daml.org/services/swsf/1.0/swso/`.

[31] S. Thakkar, C. Knoblock, and J. L. Ambite. A View Integration Approach to Dynamic Composition of Web Services. In *Proceedings of ICAPS'03 Workshop on Planning for Web Services*, Trento, Italy, June 2003.

[32] P. Traverso and M. Pistore. Automated Composition of Semantic Web Services into Executable Processes. In *International Semantic Web Conference*, pages 380–394, 2004.

[33] W. M. P. van der Aalst. Pi calculus versus Petri nets: Let us eat humble pie rather than further inflate the Pi hype, 2004. Available from `http://tmitwww.tm.tue.nl/staff/wvdaalst/pi-hype.pdf`.

[34] W. M. P. van der Aalst and A. H. M. ter Hofstede. YAWL: Yet Another Workflow Language. *Inf. Syst.*, 30(4):245–275, 2005.

[35] W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow Patterns. *Distrib. Parallel Databases*, 14(1):5–51, 2003.

[36] F.-Y. Wang, Y. Gao, and M. Zhou. A Modified Reachability Tree Approach to Analysis of Unbounded Petri Nets. *IEEE Transactions on Systems, Man and Cybernetics – Part B*, 34(1):303–308, 2004.

[37] P. Wohed, W. M. P. van der Aalst, M. Dumas, and A. H. M. ter Hofstede. 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 *Lecture Notes in Computer Science*, pages 200–215. Springer, 2003.

[38] H.-M. Wong and M. Zhou. Automated generation of modified reachability trees for Petri nets. In *Regional Control Conference, Brooklyn, NY, July 24-25, 1992*, pages 119–121, 1992.

[39] WSCDL Coalition. Web Services Choreography Description Language Version 1.0. `http://www.w3.org/TR/ws-cdl-10/`.

[40] WSDL Coalition. Web Service Description Language (WSDL) version 1.1. `http://www.w3.org/TR/wsdl`.

[41] WSMO Coalition. Web Service Modeling Ontology (WSMO) D2v1.2. `http://www.wsmo.org/TR/d2/v1.2/`.

[42] D. Wu, E. Sirin, B. Parsia, J. Hendler, and D. Nau. Automatic web services composition using SHOP2. In *Proceedings of Planning for Web Services Workshop in ICAPS 2003*, Trento, Italy, June 2003.

[43] M. T. Wynn, D. Edmond, W. M. P. van der Aalst, and A. H. M. ter Hofstede. Achieving a General, Formal and Decidable Approach to the OR-Join in Workflow Using Reset Nets. In G. Ciardo and P. Darondeau, editors, *ICATPN*, volume 3536 of *Lecture Notes in Computer Science*, pages 423–443. Springer, 2005.

[44] J. Yang and M. P. Papazoglou. Service components for managing the life-cycle of service compositions. *Inf. Syst.*, 29(2):97–125, 2004.

[45] P. Yendluri. Web services choreography. `http://www.mywebservices.org/index.php/article/articlestatic/1178/1/24/`.