Towards Semi-automated Workflow-Based Aggregation of Web Services

Antonio Brogi and Razvan Popescu

Computer Science Department, University of Pisa, Italy

Abstract. Service aggregation is one of the main issues in the emerging area of service-oriented computing. The aim of this paper is to contribute to the long-term objective of lifting service aggregation from manual hand-crafting to a semi-automated engineered process. We present a methodology which, given a set of service contracts, tries to construct an aggregation of such services. Service contracts include a description of the service behaviour (expressed by a YAWL workflow), as well as an (ontology-annotated) signature. The core aggregation process basically performs a control-flow and an (ontology-aware) data-flow analysis of a set of YAWL workflows to build the contract of an aggregated service.

1 Introduction

Service-oriented computing [18] is emerging as a new promising computing paradigm that centres on the notion of *service* as the fundamental element for developing software applications. In this scenario, two prominent issues involved in the development of next generation distributed software applications can be roughly synthesised as: (1) discovering available services that can be exploited to build a needed application, and (2) suitably aggregating such services to achieve the desired result. A typical example [16] of the need of aggregating services is a client wishing to make all the arrangements necessary for a trip (flights, hotel, rent-a-car, and so on). Such a client query may not be satisfied by a single service, while it could be satisfied by composing several services. Complex Web service interactions however require more than SOAP, WSDL and UDDI can offer [7], and semi-automatic aggregation frameworks based on such standards are not available yet.

The aim of this paper is to contribute to the long-term objective of lifting service aggregation from manual hand-crafting to a semi-automated engineered process. We present a methodology which, given a set of service contracts, tries to construct an aggregation of such services. Service contracts include a description of the service behaviour (expressed by a YAWL [23] workflow), as well as an (ontology-annotated) signature. The core aggregation process basically performs a control-flow and an (ontology-aware) data-flow analysis of a set of YAWL workflows to build the contract of an aggregated service. Technically, these analyses are defined by first expanding the services' workflows with dummy YAWL flow constructs, and by exploiting ontology-matching mechanisms to perform

B. Benatallah, F. Casati, and P. Traverso (Eds.): ICSOC 2005, LNCS 3826, pp. 214–227, 2005.
© Springer-Verlag Berlin Heidelberg 2005

a semantics-aware data-flow analysis. It is worth noting that the aggregation process is parametric with respect to the type of semantic annotations and the matching mechanism. Namely different ontology-matching mechanisms can be plugged-in (e.g., [4, 5, 17]), including the "void" one for syntactic matching (matching=identity) in absence of ontological information. The result of the aggregation process is a YAWL workflow which describes the interplay among all the services considered, namely all the control-flow and data-flow relationships among them.

In this paper we will try to focus on the aggregation process, and directly consider the problem of how to aggregate a given set of service contracts. We will not describe here how service contracts can be generated from service implementations. (A thorough analysis of how to transform BPEL [3] specifications into workflows can be found in [26].) We will not describe either how the initial set of services is chosen. We may assume that it has been selected by some matchmaking algorithm in response to some client query. For instance, the composition-oriented matchmaking algorithm in [4] returns a candidate set of services which may collectively satisfy a client query. It is worth observing that the aggregation process is completely separated from the process of selecting the initial set of services. For instance, the latter can be also performed by a user browsing a (semantics-enabled) UDDI registry and selecting some services.

It is worth noting that the proposed aggregation process can accept both black-box and glass-box queries to drive the aggregation. Black-box queries simply specify the sets of inputs and outputs that the aggregated service should request and offer respectively. Glass-box queries specify instead a process behaviour (i.e., a workflow and not just inputs/outputs) and can be used to check whether it can be aggregated together with a given set of services.

The description of the proposed aggregation process can by synthesised in three main steps: (1) perform control-flow and data-flow analysis on the input services to determine their aggregation, (2) generate the contract of the aggregated service, (3) deploy the aggregated service. We will concentrate on steps (1) and (2) in this paper, and it is worth stressing the importance of separating the phase of contract generation from the deployment of the aggregated service, thus allowing multiple deployments of the latter.

2 Aggregation Framework

2.1 Service Contracts

We consider services that are described by *contracts* [13], and we argue that contracts should in general include different types of information: (a) **Ontology-annotated signatures**, (b) **Behaviour**, and (c) **Extra-functional properties**. Following [16], we argue that WSDL signatures should be enriched with ontological information (e.g., expressed with OWL [10] or WSDL-S [15]) to describe the semantics of services, necessary to automatise the process of overcoming signature mismatches as well as service discovery and composition. Still, the

information provided by ontology-annotated signatures is necessary but *not* sufficient to ensure a correct inter-operation of services. Following [13], 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. We argue that YAWL [23] (see below) is a good candidate to express service behaviour as it has a well-defined formal semantics and it supports a number of workflow patterns. Finally, we argue that service contracts should expose, besides annotated signatures and behaviour, also so-called extra-functional properties, such as performance, reliability, or security. (We will not however consider these properties in this work, and leave their inclusion into the aggregation framework as future work.)

We intend to build an aggregation framework capable of translating the behaviour of a service described using existing process/workflow modelling languages (e.g., BPEL, OWL-S [16], etc.) into equivalent descriptions expressed through an abstract language with a well-defined formal semantics, and vice-versa. An immediate advantage of using such an abstract 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 consider that YAWL [23] 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, which 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).[1] 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 [22], 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 [22]. 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.

2.2 Aggregation Phases

As mentioned in the Introduction, a prerequisite of our framework is the set of services to be aggregated which may be obtained either by manual selection or as

[1] Space limitations do not allow us to illustrate these patterns. A thorough description of them may be found in [24].

output of a service discovery framework. It is worth noting that our aggregation approach copes both with black-box and glass-box queries. On the one hand, a black-box query specified only in terms of offered inputs and requested outputs is transformed into an equivalent service which is then added to the registry of matched services. On the other hand, one may submit services as glass-box queries. By doing so one may also check whether the corresponding service can be aggregated with a given set of services.

The semi-automated aggregation framework 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 YAWL as an abstract workflow language for expressing behaviour, and OWL for example for expressing the semantic information. One may note that such a translation may be done off-line and hence it is not a burden for the aggregation process. (A thorough analysis of how to transform BPEL specifications into workflow patterns can be found in [26].)
1. **Core Aggregation**. During this phase YAWL processes are expanded with explicit data- and control-flow (dummy) constructs, also called Input/ Output Control/Data enabler processes (or ICs/IDs/OCs/ODs for short). We then express the initial control-flow connections in terms of the newly added ICs and OCs. Next, we use data-flow dependencies (i.e., operation and message mapping among the involved parties) provided by an ontology-aware matching algorithm (e.g., [4, 5, 17]) to derive a data-flow mapping. We express such mapping by suitably linking IDs and ODs.
2. **Contract Generation**. Firstly, we perform a basic check to see whether the aggregated service does not have processes with unsatisfied inputs. Should this be the case, we adequately eliminate unlinked ODs and other redundant dummies introduced by the previous phase, and we cancel redundant control-flow constructs. The ontology-annotated signature and behaviour we obtain form the service contract of the aggregated service. The generated contract can be further analysed (e.g., lock analysis) and optimised.
3. **Service Deployment**. Finally, the aggregated service can be deployed as a real-world Web service (i.e., described using OWL-S, or BPEL + semantics, etc.). Clients will hence see the aggregation as another Web service that can now be discovered and further aggregated with other services. This operation is the inverse of the operation done during the Service Translation phase.

As already mentioned in the Introduction, we will describe phases (1) and (2) in the following, after introducing some definitions.

2.3 Definitions

We shall use the term "service" to denote the YAWL notion of "workflow specification", "process" to denote a YAWL "task" as well as "start" and "end" to denote YAWL "input condition" and "output condition", respectively.

We consider a set or registry of service contracts to be aggregated, where each contract corresponds to an original service implementation (e.g., described with BPEL and OWL for semantics, etc.). A contract S consists of an ontology-annotated signature (i.e., semantic information, *Sem* for short) and of a behaviour description (*Beh*).[2]

Sem specifies the set of processes (*Procs*) as well as the name (*Sname*) and the type (*Stype*) of the service. Indeed we argue that services, processes as well as parameters (i.e., messages) should be annotated with ontological information describing their types. Such information can be used by discovery frameworks to better match services. For example, considering ontologies for services, processes and parameters, we may have for example a "stock_quote" service type, a "flight_reservation" process type, or a "notebook_computer" parameter type, and so on. *Procs* consists of the m processes of S together with *start* and *end*, which are two special dummy processes used to mark the entry end exit points, respectively, of S. A process P contains the sets of input (I) and output (O) parameters, its name (*Pname*) and type (*Ptype*). Similarly to services and processes, a parameter exposes its name (*Iname*) and type (*Itype*). Note that the matching concerns types — rather than names — of parameters, processes or services[3]. Name matching should be employed in absence of ontology-annotations. The *start* and *end* dummy processes are defined similarly to the other processes P yet they do not have IOs and ontological values associated. They are named "$DummyStart\ of\ P$" and "$DummyEnd\ of\ P$", respectively.

Beh contains information about both the control-flow constructs used by processes in *Procs* (PC), as well as information about the control-flow dependencies among such processes (PD). PC associates one join and one split construct to each process P. A join or split control construct may be one of the following: AND, OR, XOR, or EMPTY. Intuitively, the join specifies "how many" processes before P are to be terminated in order to execute P, while the split construct specifies "how many" processes following P are to be executed. The EMPTY join (split) is used when *at most one* process execution precedes (follows, respectively) the execution of P. PD defines the control-flow of S by means of a set of process pairs. A pair $< P, Q >$ specifies that P must be executed before process Q (i.e., Q may begin its execution provided P has finished its execution).

Consider the following example which will be used as a basis for presenting the applicability of our methodology, and for enhancing the description of the proposed approach. A youngster passionate about winter sports and computer science, decides to publish on her homepage a Web service providing information on the conditions of her favourite slope. Basically, she wishes that other winter

[2] When necessary, indexes shall be used for disambiguation.

[3] Roughly, service matching may restrict the set of services to be considered, while process matching may help refining further the selection (e.g., matching a "computer_selling" process of an "e_shop" service) to possibly aggregate sub-services rather than whole services. Finally, parameter matching can provide the data-flow information necessary to achieve the aggregation.

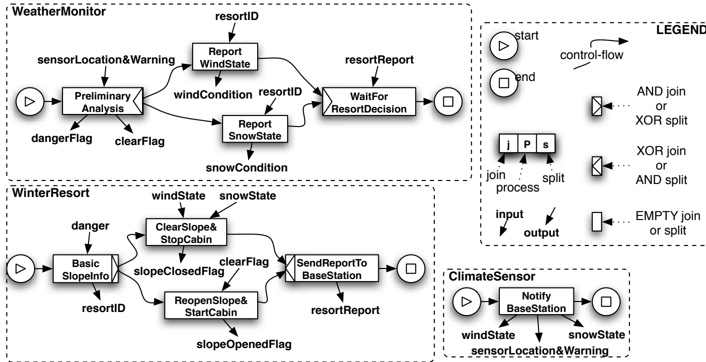

Fig. 1. Example registry with three services to be aggregated

sports enthusiasts like her may access her page in order to see whether the slope is practicable and the cabin is working.

One may assume that she locates the ontologically-enriched WR[4] service (see Figure 1[5]) from a (semantically anotated) UDDI registry. Next, she feeds this service as a black-box query to a discovery framework (e.g., [9] or [5]). This may lead to selecting the other two services in Figure 1.

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 persons with different programming styles and backgrounds may present (aggregation) issues. It is likely that the selected services do not match perfectly, or that the ensemble is not optimal, and so on. Redundancies (e.g., redundant outputs) may occur as well. The three services are as follows:

CS basically gathers data from sensors located on top of the mountain. Upon invocation, it executes process NBS which outputs the sensor's location and the warning level for the slope it is monitoring sLW, as well as the snow's condition sS (e.g., indication of avalanche danger) and the wind's condition wS (e.g., strong wind leads to stopping the cabin). We may assume that CS runs periodically (e.g., every hour).

WM (or $BaseStation$) centralises data gathered from various CSs. It firstly performs a preliminary analysis (e.g., reasoning based on a history record over the past X years) through the execution of PA. On the one hand it specifies whether there is an avalanche danger by enabling dF or, on the other hand whether the slope is safe (e.g., it may be (re)opened). In the latter case cF is enabled. The AND split of PA indicates that *both* RWS and RSS are to be executed after it. RWS makes its own prediction on the wind state based on the rID input. Similarly, RSS sets the snow state based on its prediction. The AND join construct of WRD states that WRD may be executed provided *both* RWS

[4] Due to space limitations, we shall use abbreviations throughout the paper (e.g., WR instead of $WinterResort$).

[5] In addition to the representation of YAWL tasks (i.e., processes) we graphically describe their parameters as well.

and RSS finished execution. WRD is in charge of waiting for a report from a WR service (i.e., the decision of the latter on whether to close or to (re)open the slope).

WR is a service that manages access to a slope and cabin. From a workflow point of view, WR behaves differently from WM in the way that it uses a XOR split in the BSI process and a XOR join in the $SRBS$ process. The former indicates that *either* $CSSC$ *or* $RSSC$ will be activated for execution, while the latter indicates that $SRBS$ will be invoked after each execution of *either* $CSSC$ *or* $RSSC$. BSI inputs the danger flag produced by the WM and it decides either to clear the slope and stop the cabin (by executing the $CSSC$ process), or to (re)open the slope and (re)start the cabin (by executing the $RSSC$ process). Finally, $SRBS$ sends a report to the WM service with its decision.

2.4 Core Aggregation

During this phase, all processes (except *start* and *end* ones) are expanded with explicit control- and data-flow dummies. Then, a control-flow analysis expresses the initial flow dependencies in terms of the newly added dummies. Last but not least, a data-flow analysis coordinates processes of (possibly) different services by taking into account a given data-flow mapping. The three steps are detailed hereafter.

PROCESS EXPANSION

Let us consider the empty (aggregated) service A. For each process P of each service S, we generate the following five dummy processes:

- P^* corresponding to process P "stripped off" its join and split control constructs, and augmented with AND join and split constructs,
- an Input Control enabler IC_P which inherits the initial join of P,
- an Output Control enabler OC_P which inherits the initial split of P,
- an Input Data enabler ID_P which is in charge of gathering all inputs needed for the execution of P (if P has at least one input), as well as
- an Output Data enabler OD_P which "offers" all outputs of P to other processes (if P has at least one output).

With the exception of P^*, all such processes lack IOs and ontological values. Their purpose is to explicitly separate the control- and data-flow logic of P. From a control-flow point of view, IC_P and ID_P are linked as inputs of P^* while OC_P and OD_P are linked as outputs. All added dummies as well as the corresponding dependencies have to be added to Beh_A.

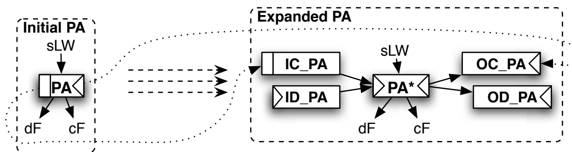

Fig. 2. Expansion of PA

Figure 2 describes the process expansion step applied to process PA of service WM. As one may note, PA^* employs AND join and split constructs as, on the one hand, both IC_PA and ID_PA have to finish execution before executing PA^* and, on the other hand, both OC_PA and OD_PA are to be executed after PA^* terminates. From a data-flow point of view, the AND join of ID_PA indicates that all inputs of PA must be available in order to execute PA. Dually, the AND split of OD_PA specifies that after PA finishes its execution, all its outputs will be available to all processes requesting at least one of them as input.

Once all processes have been expanded, two more processes 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 ICs of all services to be aggregated. Dually, OC_A has an AND join in order to wait for OCs of all services to finish execution. Links from *start_A* to IC_A as well as from OC_A to *end_A* are added to Beh_A.

Control-Flow Analysis

During this step, control-flow dependencies of each service S are specified in terms of the newly added ICs and OCs, as well as IC_A and OC_A, and then added to Beh_A. The result of applying this step on the WM service may be seen in Figure 3.[6]

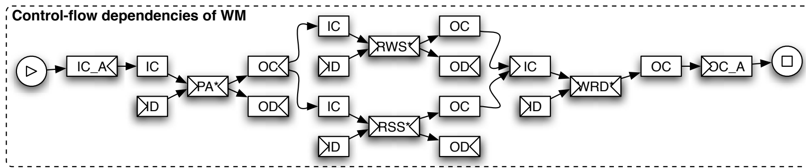

Fig. 3. Control-flow analysis for WM

For example, the initial link between PA and RWS has been translated to a link between OC_PA and IC_RWS. Moreover, one should note that *start_WM* and *end_WM* are now connected to IC_A and OC_A respectively. That is, IC_A enables (from the control-flow point of view) IC_PA for execution. Dually, OC_WRD is connected to OC_A and hence (from the control-flow point of view) its execution is to be interpreted as the termination of WM.

Data-Flow Analysis

In order to derive data-flow information linking processes of (possibly) different services, one has to match requested inputs with offered outputs. Our flexible methodology allows for an ontology-based matching algorithm (e.g., [17, 5]) to be plugged-in. "An input i of process P matches an output o of process Q if and only if $Itype_i$ is in an *exact* or *subsumes* relation with $Otype_o$". Dually, "an output o of process Q matches an input i of process P if and only if $Otype_o$ is in an *exact* or *plug-in* relation with $Itype_i$". One should note that the notion of

[6] All enabler dummies shall be abbreviated in figures from now onwards (e.g., IC instead of IC_PA, and so on).

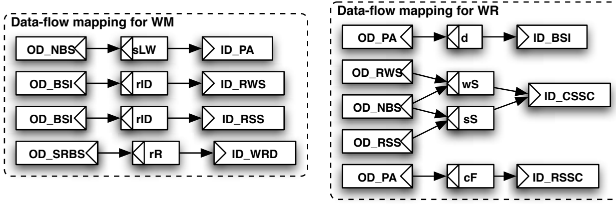

Fig. 4. Data-flow analysis for our example

"match" used in this paper is in line with the one defined in [17, 16]. We shall call such a match a *data-flow dependency* and a set of them as *data-flow mapping*.

From a data-flow point of view, a process P must have all its inputs available in order to be executable. In this paper we assume that such data-flow dependencies are provided by the matching framework. A maximal such mapping can be obtained by employing a one-to-one matching between all process parameters of the services to be aggregated. One should note that the user should be allowed to modify, cancel or add dependencies in the mapping. A data-flow mapping can be expressed in terms of IDs and ODs as follows. If input x of process P matches output(s) y of process(es) Q then we generate the following:

1. A dummy process P_x[7] with no IOs or ontological value. However, it is important to note that such a dummy employs a XOR join and an EMPTY split. This is due to the fact that values for x may be obtained from different ys, yet only one is needed. Furthermore, a link from P_x to ID_P is added to Beh_A.
2. A link from OD_Q to P_x which is added to Beh_A for every matched y.

Figure 4 illustrates the data-flow mapping for our example. Due to space issues, P_x dummy names will be abbreviated to x in figures from now onwards. One should note that the CS service is not depicted as its only process (NBS) does not have inputs.

2.5 Contract Generation

During this phase, the algorithm employs an input-driven basic check and then it cleans the aggregated service A of redundant constructs.

Basic Validation

We firstly assume that all services are "well defined" in the sense that each initial process P has at least one incoming link (with the exception of "*start*") and at least one outgoing link (with the exception of "*end*"). This means that each IC has at least one incoming link, and that each OC has at least one outgoing link. At this point one may encounter two situations:

– All processes P have their inputs satisfied. In other words, every input x of P has been matched with at least one output y of a process Q. This

[7] For simplicity we assume here that all P_x are unique.

translates to the fact that the P_x dummy process has at least one incoming link. Should this be the case, we say that the aggregation is successful — in the way that there are no unsatisfied data- (and control-) flow constraints.

– At least one process P is missing some inputs. In other words there exists an input x of P which has not been matched to any output(s) y of process(es) Q. This translates to the fact that the P_x dummy process has no incoming links. Should this be the case, we say that the aggregation has not succeeded — in the way that there is at least one unsatisfied data-flow constraint. The (additional) missing inputs must be provided by other services, hence either a refined query can be launched or the needed services can be manually added to the set of services to be aggregated.

We chose to consider as valid such a "closed" workflow (i.e., without unsatisfied inputs) in order to enforce a necessary yet not sufficient condition for the execution of (all) processes. Given a valid service contract, one may use analysis tools in order to verify (dead-)lock freedom for example. As YAWL is built upon Petri Nets (PN), analysis tools for the latter can be exploited to check properties of PN translations of the former. For example WofYAWL [25] is an analysis tool for YAWL workflows. WofYAWL maps an input YAWL workflow into a PN with inhibitor arcs, and then analyses semi-positive transitions in the short-circuited net. If the net is bound, it performs a relaxed soundness check in the regular net. Finally the results are mapped back into a YAWL workflow, possibly annotating the output with warnings (e.g., in the case of unbounded nets). Figure 5 describes the aggregation contract we have obtained so far for our example.[8] One should note that all ICs (e.g., IC_PA and so on) have at least one incoming link, as well as, all OCs (e.g., OC_PA and so on) have at least one outgoing link. Moreover, all process with the exception of OD_CSSC and OD_RSSC have at least one incoming and one outgoing link. We can say that the aggregation is successful as there are no unsatisfied data- (or control-) flow constraints.

Eliminating Redundancies

As one may have noted, not all dummy constructs introduced during the **Core Aggregation** phase are necessary. Given the aggregated service is valid, we can (repeatedly) eliminate redundant items, that is, dummies and join/split constructs. One obtains at the end of this step the final service contract of A. We hereafter describe three elimination criteria.

Dummy Absorption. Assume a dummy (i.e., control or data process enabler, or process added during the data-flow analysis) iD connected as input of another process P such that the pair $< join_{iD}, join_P >$ is one of the following – $\{< EMPTY, EMPTY >, < EMPTY, \alpha >, < \alpha, \alpha >\}$ –, where $\alpha \in \{AND, XOR, OR\}$. Then, we "absorb" iD into P which remains unchanged. If $< join_{iD}, join_P >$ is $< \alpha, EMPTY >$ then we absorb iD into P with the observation that P inherits the join of iD (i.e., $join_P := join_{iD}$). The scenario is dual

[8] Due to its verbosity we chose not to represent dummies introduced during the Data-Flow step – with the exception of wS and sS. Moreover, the full graphical form of the workflow (i.e., including process parameters and so on) has been omitted.

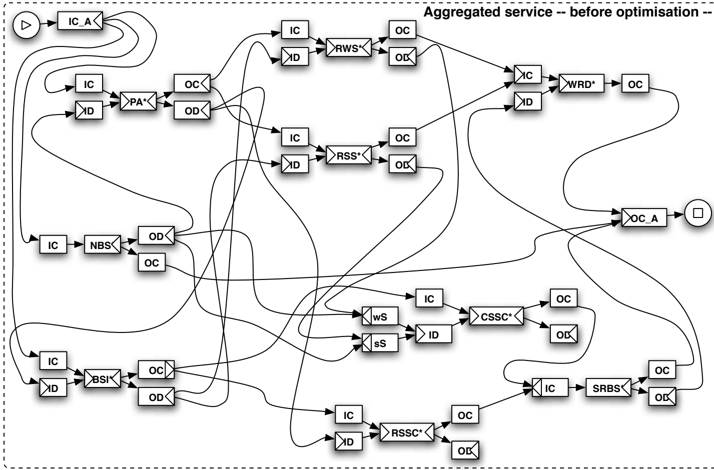

Fig. 5. Service contract A (before eliminating redundancies)

for absorbing output dummies. Absorbing means eliminating iD and updating Beh_A correspondingly.

Dummy Elimination. An OD_P employing an EMPTY split construct and that does *not* have at least one outgoing link to other join of an ID_Q can be eliminated together with its corresponding link (from P to OD_P) from Beh_A. One should note that the initial AND split of OD_P should be cancelled first by the following criteria.

Join/Split Elimination. A $join_P \neq$ EMPTY has to be set to EMPTY provided P has *only one* incoming link. The dual (i.e., the "reset" of $split_P$ given P has *at most one* outgoing link) is resolved in similar way.

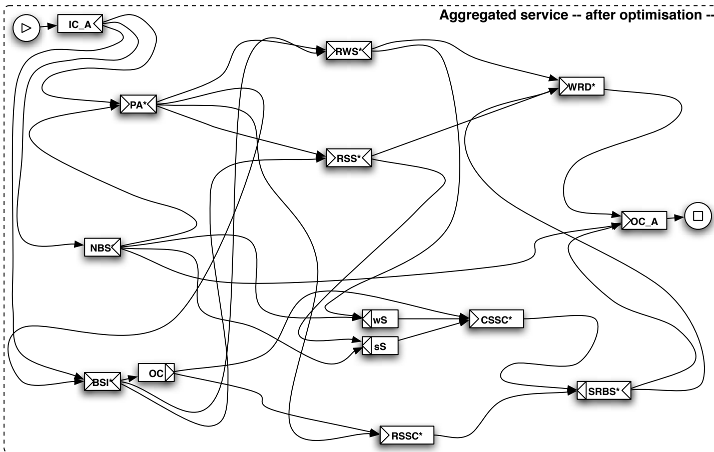

Fig. 6. Final service contract A

Let us come back to our example. Figure 4 indicates that all dummies introduced during the data-flow analysis are redundant, except for wS (input of $CSSC$) and sS (input of $RSSC$). The redundant joins are cancelled first and then the respective redundant processes are absorbed into IDs and ODs. Moreover, the elimination criteria allow us to cancel almost all dummies introduced during Process Expansion with the exception of OD_CSSC, OD_RSSC, and OC_BSI. The former two are tackled by the dummy elimination criterion. The final version of A is given in Figure 6.

3 Concluding Remarks

The aim of this paper is to contribute to the long-term objective of lifting service aggregation from manual handcrafting to a semi-automated engineered process. We have presented the kernel of a semi-automated workflow based aggregation framework of Web services. It consists of a methodology which, given a set of service contracts, tries to construct an aggregation of such services.

We have synthesised three main phases of the proposed aggregation process: (1) *Core Aggregation* – perform control- and data-flow analysis on the input services to determine their aggregation, (2) *Contract Generation* – generate the contract of the aggregated service, (3) *Service Deployment* – deploy an implementation of the aggregated service. While we concentrated on steps (1) and (2) in this paper, it is worth stressing the importance of separating the phase of contract generation from the deployment of the aggregated service, which allows multiple deployments of the latter.

The main features of our approach are: (a) It can be used to aggregate services written with different description languages (e.g., BPEL + semantics, OWL-S), (b) It is (semi-)automatic – both with respect to service translation and coordination (core aggregation and contract generation), (c) It allows a seamless integration with service discovery systems (third-party matchmaking frameworks can be straightforwardly plugged in), (d) It supports both black- and glass-box queries (i.e., behaviour-less and behavioural queries), (e) It features compositional aggregation (e.g., the aggregation of A, B, and C can be computed by first aggregating A and B and then aggregating the obtained service with C), and finally (f) It supports multiple deployments of the aggregated service.

Regrettably, space limitations do not allow a thorough discussion of related work (e.g., manual [3, 28], semiautomatic [9, 12] or fully automatic approaches [2, 11, 19, 20, 21, 27]). Surveys on Web service composition can be found in [1, 6, 8, 14]. In manual Web service composition the requester acts as the service composer as well. She has to browse the registry, find the desired service operations and model their interactions into a flow structure. Fully automatic composition of services is very difficult to achieve as the requester has to specify all input requirements of registered service operations that make the composite service. Furthermore, processing the request is a very time consuming process. A significant number of fully automatic approaches employ planning techniques. A downside of planning is that both the goal and the status are difficult to represent. Another issue is that all ser-

vices involved in the composition have to be known a priori. It is however worth observing that, while some of the previously mentioned features ((a) – (f)) are considered in some existing approaches, our approach is the first — at the best of our knowledge — that provides all of them in a single framework.

A key ingredient of our framework is the notion of service contract, which includes a description of the service's behaviour (expressed by a YAWL [23] workflow), as well as an (ontology-annotated) signature. 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 aggregrating services written in different languages and for multiple deployments of the aggregated service.

Further investigation will be devoted to extend the core aggregation process in order to ensure stronger formal properties of computed aggregations, and to account for the adaptation of signature and behavioural mismatches in contracts. Future work will also be devoted to the development of the semi-automated derivation of contracts from real service implementations (considering first BPEL and OWL-S, and exploiting the techniques described in [26]), and of the service deployment phase (again considering BPEL and OWL-S first).

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. 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.
3. BPEL4WS Coalition. Business Process Execution Language for Web Services (BPEL4WS), 2002. http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/.
4. A. Brogi, S. Corfini, and R. Popescu. Flexible Matchmaking of Web Services Using DAML-S Ontologies. In P. Traverso and S. Weerawarana, editors, *Proceedings of Second International Conference on Service Oriented Computing (ICSOC04 - short papers), IBM Research Report. NY, USA*, pages 30–45, November 15-18 2004.
5. 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.
6. Y. Charif and N. Sabouret. An Overview of Semantic Web Services Composition Approaches. To appear in Proceedings of the International Workshop on Context for Web Services 2005, Elsevier.
7. H.-P. Company. Web Services Concepts – a technical overview. http://www.hpmiddleware.com/downloads/pdf/web_services_tech_overview.pdf. Technical report, 2001.
8. J. Koehler and B. Srivastava. Web Service Composition: Current Solutions and Open Problems. ICAPS Workshop on Planning for Web Services, pp. 28-35, 2003.
9. 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.

10. D. McGuiness and F. van Harmelen (Eds). OWL Web Ontology Language Overview. Web guide, February 2004. http://www.w3.org/TR/owl-features.
11. S. McIlraith and C. T. Son. Adapting Golog for composition of semantic Web services. Proceeding of 8th Conference on Knowledge Representation and Reasoning (KR'02), 2002.
12. B. Medjahed, A. Bouguettaya, and A. K. Elmagarmid. Composing Web services on the Semantic Web. *The VLDB Journal*, 12(4):333–351, 2003.
13. L. Meredith and S. Bjorg. Contracts and types. *CACM*, 46(10), 2003.
14. N. Milanovic and M. Malek. Current Solutions for Web Service Composition. *IEEE Internet Computing Online*, 8(6):51–59, Dec. 2004.
15. J. Miller, K. Verma, P. Rajasekaran, A. Sheth, R. Aggarwal, and K. Sivashanmugam. WSDL-S: Adding Semantics to WSDL - White Paper. http://lsdis.cs.uga.edu/library/download/wsdl-s.pdf.
16. OWL-S Coalition. OWL-S 1.1 release. http://www.daml.org/services/owl-s/1.1/.
17. 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.
18. M. P. Papazoglou and D. Georgakopoulos. Service-Oriented Computing. *Commun. ACM*, 46(10):24–28, 2003.
19. R. Ponnekanti and A. Fox. SWORD: A developer toolkit for building composite Web services. Computer Science Department, StanfordUniversity, 2002. http://www2002.orgCDROM/alternate/786/.
20. S. Thakkar, A. C. Knoblock, and L. Ambite. A view integration approach to dynamic composition of Web services. Proceedings of the ICAPS '03 Workshop on Planning for Web Services, Italy, 2003.
21. P. Traverso and M. Pistore. Automated Composition of Semantic Web Services into Executable Processes. In *International Semantic Web Conference*, pages 380–394, 2004.
22. 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.
23. W. M. P. van der Aalst and A. H. M. ter Hofstede. YAWL: Yet Another Workflow Language. Technical report, Queensland Univ. of Technology, FIT-TR-2003-04, 2003.
24. 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.
25. E. Verbeek. WofYAWL Version 0.3. Technical report available online at http://home.tm.tue.nl/hverbeek/wofyawl03.pdf.
26. 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.
27. D. Wu, E. Sirin, J. Hendler, D. Nau, and B. Parsia. Automatic Web services composition using SHOP2. Proceedings of the ICAPS '03 Workshop on Planning for Web Services (P4WS '03), 2003.
28. J. Yang and M. P. Papazoglou. Service components for managing the life-cycle of service compositions. *Information Systems*, 29(2):97–125, 2004.