Università di Pisa

Dipartimento di Informatica

Technical Report: TR-06-20

Design and Implementation of Sator: a Web Service Aggregator

Antonio Brogi Razvan Popescu

Matteo Tanca

Computer Science Department, University of Pisa, Italy

December 21, 2006

Address: Largo B. Pontecorvo 3, 56127 Pisa, Italy. Tel: +39 050 2212700 Fax: +39 050 2212726

Design and Implementation of `Sator`: a Web Service Aggregator

Antonio Brogi Razvan Popescu

Matteo Tanca

Computer Science Department*, University of Pisa, Italy

December 21, 2006

Abstract

The aggregation methodology we propose in this paper automatically generates the *service contract* of a composite service from a set of contracts to be aggregated together with a data-flow mapping linking service parameters. Service contracts consist of (WSDL) signature, (OWL) ontology information, and (YAWL) behaviour specification.

The aggregation process generates the workflow of the composite from the initial workflows by suitably adding control-flow constraints among their tasks due to data-flow dependencies among task parameters.

After describing the whole methodology, we will also give an insight on our proof-of-concept Java prototype implementation of the aggregation process.

1 Introduction

Service-oriented Computing [21] aims at building future heterogeneous, distributed business applications through the use of (Web) services as building

*Largo B. Pontecorvo 3, Pisa, 56127, Italy. Emails: {brogi|popescu}@di.unipi.it, tanca@cli.di.unipi.it

blocks. Currently, WSDL [34] interfaces provide only a syntactic description of services similar to IDL interfaces for software components. Consequently, the generation of composite services[1] from black-box (viz., behaviour-less) service descriptions may (dead)lock. BPEL [5] is the main proposal for composing services, and it is highly promoted by the industry, yet the designer is in charge of manually selecting the services (e.g., from UDDI [27] registries) and generating the composite one. The Semantic Web initiative proposes ontology-aware languages such as OWL-S [19] to automate Web service discovery, composition and monitoring. Basically, the ontology information can be used to (automatically) match service parameters, and such matches can be exploited to enhance not only service discovery but service composition and adaptation as well. Most existing automation-oriented approaches employ A.I. techniques such as planning (e.g., [4, 16, 26, 36]), still the goal is difficult to represent and the aggregation process is quite time-consuming. Furthermore, the abundance of languages to express service compositions [5, 19, 33, 32] obstructs the achievement of automated Web service aggregation, as currently, 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 (viz., ontology) and behavioural mismatches in view of application integration within and across organisational boundaries.

In this paper we present `Sator`, a (Web) service aggregator that, given a set of advertised service contracts together with a data-flow mapping linking service parameters, automatically generates the contract of a composite service. Service contracts include (WSDL [34]) signature, (OWL [15]) ontology information, as well as (YAWL [29]) behaviour specification. The aggregation process generates the workflow of the composite from the initial workflows by suitably adding control-flow constraints among their tasks due to data-flow dependencies among

[1] We shall use the terms "composition" and "aggregation" interchangeably throughout the paper.

parameters. 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.

To the best of our knowledge our methodology is the first one to offer the following features:

- it is amenable to efficient implementations, as it relies on service contracts, which can be generated off-line,
- it can be employed to discover [8], aggregate [8, 10], and adapt [7, 9] BPEL processes, as it straightforwardly integrates with the BPEL2YAWL translator described in [11],
- it provides the basis to discover, aggregate, and adapt 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.

The rest of the paper is organised as follows. Section 2 introduces a motivating example that will be used in Section 5 as a basis for illustrating the aggregation methodology. In Section 3 we introduce the service contracts, while in Section 4 we briefly describe YAWL. Section 5 is dedicated to the core aggregation methodology. In Section 6 we describe step-by-step some aggregation examples that illustrate how the methodology is able to cope with various YAWL constructs. Section 7 gives an insight on our proof-of-concept prototype implementation of `Sator`. In Section 8 we briefly review related work. Finally, Section 9 presents some concluding remarks.

2 Motivating Example

Figure 1 presents a simple example that we shall use throughout the paper for describing the aggregation methodology.

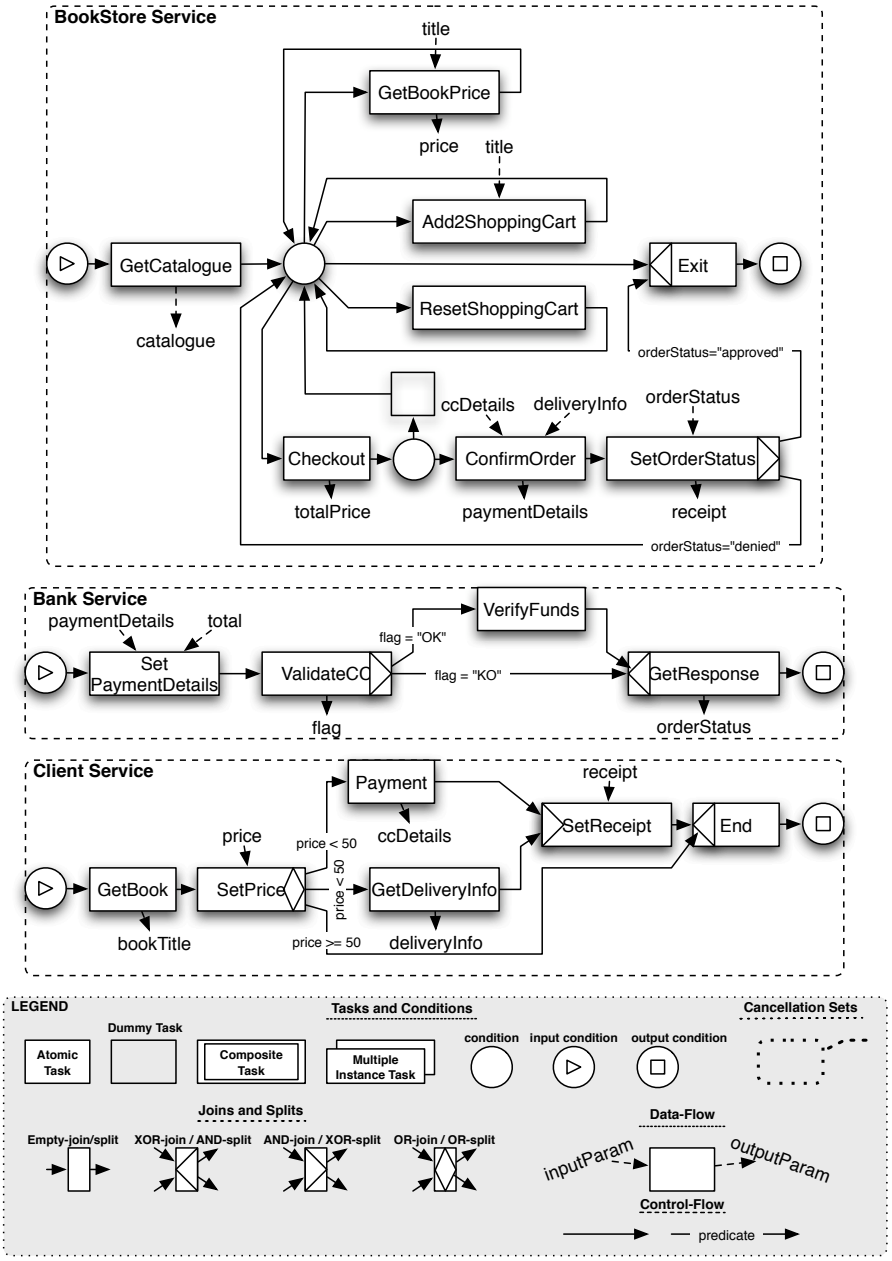

Figure 1: Example illustrating the workflows of three (interacting) Web services.

The *BookStore* workflow[2] describes the protocol of a service that sells books. When executed, the token placed in its input condition enables for execution

[2]Please note that the terms "workflow" and "service" are used interchangeably throughout the paper.

the *GetCatalogue* task, which outputs a *catalogue* value. The token placed in the following deferred choice [29] enables for execution several tasks. If the client choses to execute the *GetBookPrice* task, then the workflow inputs the *title* of a book (from the client) and it outputs its *price* (to the client). Similarly, the *Add2ShoppingCart* task inputs the title of the book the client wishes to buy, while *ResetShoppingCart* removes all items previously added to the cart. If the client does a *Checkout*, then the workflow will output a *totalPrice*, which is the cost of the books in the cart. Next, a token is placed in the deferred choice following the *Checkout* task. Now, the client has the possibility to invoke, *either* one of the *GetBookPrice*, *Add2ShoppingCart*, *ResetShoppingCart*, *Checkout*, or *Exit* tasks, *or* the *ConfirmOrder* task. Note that the execution of any of the former five tasks leads to the removal of the *ConfirmOrder* tasks from the list of tasks that can be executed by the client. This is due to the fact that their execution consumes the token in the input condition of the *ConfirmOrder* task. The *ConfirmOrder* task, whose execution has to immediately follow the execution of the *Checkout* task, inputs the credit card information (*ccDetails*), as well as the client's address used for delivery (*deliveryInfo*), and it outputs the *paymentDetails*, which (as we shall see later) are to be used by the *Bank* service to verify the validity of the transaction. The order confirmation is followed by the execution of the *SetOrderStatus* task, which inputs the response of the *Bank* service (*orderStatus*), and it outputs a *receipt* to the client. If the *Bank* approved the transaction, the execution continues with the *Exit* task which logically marks the end of a buying session. Otherwise, a token is placed into the first deferred choice. Note that the client has also the possibility of terminating the buying session by executing the *Exit* task at any moment after the execution of the *GetCatalogue* task, but while it is waiting for a *receipt* from the *BookStore*.

The second workflow in the example describes a *Bank* service that can be accessed, for example, by the *BookStore* in order to validate the credit of a book-buyer. The execution of the *Bank* workflow starts with the execution of the *SetPaymentDetails* task, which inputs the *paymentDetails* as well as the

total price to be paid by the person indicated in the *paymentDetails*. The execution continues with the *ValidateCC* task, which verifies the credit card information (e.g., the credit card number and validity period), and it outputs a *flag* that is used internally by the *Bank* workflow to determine the control-flow. A "KO" value of the *flag* indicates that the supplied credit card information is *not* valid, and the execution continues with the *GetResponse* task, which outputs to the client (i.e., invoker) of the *Bank* service a corresponding *orderStatus* response. Otherwise, an "OK" value of the *flag* leads to the execution of the *VerifyFunds* task, which checks, for example, whether the book-buyer can afford paying the books. Please note that, in order to ease the presentation, we did not represent all the task inputs and outputs (IOs), such as the *total* output of *SetPaymentDetails*, which has to be (at a later moment) inputted by the *VerifyFunds* task as well. (All such YAWL mapping details such as passing values among internal tasks of a workflow have been left out intentionally.) Finally, the execution of the *VerifyFunds* task leads to the termination of the workflow due the execution of the *GetResponse* task.

The third workflow depicts a simple *Client* service that attempts to buy a book from an e.g., *BookStore* service. At the start of the workflow, the invoker of the *Client* service executes the *GetBook* task, which outputs the title of the desired book (*bookTitle*). Next, the *SetPrice* task inputs the *price* of the respective book, and depending on its value, the execution continues with one of the following two scenarios. On the one hand, should the book price not exceed a certain amount of money (e.g., 49,99 euros), the invoker has to execute in any order she wishes the *Payment* and the *GetDeliveryInfo* tasks. The former outputs the invoker's credit card details (*ccDetails*), while the latter outputs the address where the book is to be delivered (*deliveryInfo*). In this scenario, the workflow continues with the execution of the *SetReceipt* task, which waits for a *receipt* for the book being bought, and then with the *Exit* task. On the other hand, if the book price is higher than the predefined amount, the workflow finishes with the execution of the *Exit* task.

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

It is worth noting that, given the *Client* service, there are at least two possible scenarios for selecting the *BookStore* and the *Bank* services from a registry of service (contracts) advertisements. On the one hand, one can manually browse a UDDI registry of service contracts [8], while on the other hand, one can use an ontology-aware matching algorithm for such purpose. We recall that we argue for services described by contracts that contain ontology information about the service IOs. In [8] we show how service execution traces can be derived from service contracts and then matched in order to locate services that collectively can satisfy a query represented as another service. For our example, the *Client* service can be used as a query that leads to the selection of the *BookStore* and the *Bank* services.

In this paper we assume for simplicity only *exact* matches [20] among IOs of the three services. The IO matches are illustrated in Figure 2. For example, the match between the *price* input of *Client* and the *totalPrice* output of *BookStore*

Client	BookStore	Bank
GetBook():bookTitle	GetBookPrice(title):... --- Add2ShoppingCart(title)	-
SetPrice(price)	GetBookPrice():price --- Checkout():totalPrice	SetPaymentDetails(total)
Payment():ccDetails	ConfirmOrder(ccDetails):...	-
GetDeliveryInfo():deliveryInfo	ConfirmOrder(deliveryInfo):...	-
SetReceipt(receipt)	SetOrderStatus():receipt	-
-	GetCatalogue():catalogue	-
-	ConfirmOrder():paymentDetails	SetPaymentDetails(paymentDetails)
-	SetOrderStatus(orderStatus):...	GetResponse():orderStatus
-	-	ValidateCC():flag

LEGEND

Task(matchedInput):... Task():matchedOutput

Figure 2: IO matches of the three services.

(second row in the IO matches table) leads to considering the *BookStore* service as a candidate for (collectively) satisfying (together with other matched services) the *Client* service. Furthermore, the match between the *orderStatus* input of *BookStore* and the *orderStatus* output of *Bank* leads to adding the *Bank* service to the candidates list. The candidate set containing the *BookStore* and the *Bank* service is a valid candidate set [8] because the set of inputs needed collectively by the two services, together with the *Client* one is contained in the set of outputs generated by them. Please see [8] for a detailed description of the trace-based ontology-aware service selection methodology.

Now, if we assume that e.g., the *bookTitle* and the *title* ontology concepts do *not* belong to the same ontology[3], the matching algorithm would *not* be able to automatically match them. However, the matchmaker described in [8] is able to

[3]Please note that we do not include a (partial) ontology of parameter types for the examples in this paper as the accent here is on the core aggregation process and not on matching service IOs.

match the two concepts if the client (of the aggregation methodology) provides the set {*bookTitle*, *title*} as a set of equivalent ontology concepts. We use such sets of equivalent concepts so as to cope with cross-ontology mappings. It is important to note further that the client is allowed to modify, append, and/or remove matches from the matches table. For example, the match between the *price* input of the *SetPrice* task of the *Client* service and the *price* output of the *GetBookPrice* task of the *BookStore* service should be removed by the client from the table of IO matches, as the *totalPrice* of the *Checkout* task of the *BookStore* service actually reflects the amount of money that the client has to pay for the book, as we assume a constant delivery cost which is included in the *totalPrice*.

In the following we shall describe in detail the core aggregation methodology, which given a set of service contracts together with a data-flow mapping (i.e., the table of matches among IOs of the participant services) is able to automatically generate the contract of the aggregated service. Furthermore, in Section 6, a few examples based on the one presented in this Section will be used for explaining in detail the aggregation methodology.

3 Service Contracts

Currently, providers publish (purely syntactic) WSDL [34] advertisements to UDDI [27] registries (constructed in the style of yellow pages) that in turn provide clients with keyword- or taxonomy-based service discovery capabilities. Moreover, WSDL descriptions do not include any *semantic information* and hence they are not "self-described" in a machine-interpretable way. This severely limits the quality of the discovery results as the matched services may not necessarily offer the requested functionality, and hence fully-automated service 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 [19], SWSO [24], WSMO [35], or METEOR-S [23] annotate services with semantic information. BPEL [5], WSCI [33], WSCDL [32], METEOR-S [2], OWL-S [19], SWSO [24], or recently YAWL [29] 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 [5], OWL-S [19], etc.), as well as to be able to have 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 *ontology information* and/or *formal semantics*.

As a consequence, in order to tackle these two issues we consider services that are described by *contracts* [18], 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 services. Following [19], we argue that (WSDL) signatures should be enriched with ontology information (e.g., expressed with OWL [15] 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 discovery and composition. Still, the information provided by the signature and ontology information 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 its 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 [18], 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 consider that YAWL [29] is a promising candidate to be used as an abstract language for describing the 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. In the following Section we briefly present YAWL and motivate the choice of YAWL as an intermediate language.

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 methodology as future work.)

4 Background: Yet Another Workflow Language (YAWL)

An informal description of YAWL workflows [29] has been given in the Section 2 through the illustration of a few example workflows. As previously mentioned, in this Section we briefly describe YAWL by presenting some insights on the key elements and features of the language.

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 [30]. 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 [28], 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 [28]. 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. (YAWL tasks and conditions can be interpreted as Petri net transitions and places, respectively [29].) Each EWF-net has a single *input condition* and a single *output condition*. For example, all the workflow specifications depicted in Figure 1 consist of a single EWF-net.

A YAWL task may be either *atomic* or *composite*. An atomic task (e.g., *GetCatalogue*, *GetBookPrice*, and so on in Figure 1) corresponds to a leaf of

the tree. A composite task corresponds to a EWF-net at a lower level in the hierarchy. The EWF-net without any composite tasks referring to it is called *top-level workflow* and it corresponds to the root of the tree-like hierarchy. The *ExecutePayment* task of the *Bank2* workflow in Figure 8 is a composite task, which expands to an EWF-net consisting of four atomic tasks. A task can have multiple instances that can be created either statically or dynamically. Lower and upper bounds are used to specify the number of instances that can be created. Furthermore, a threshold value may be used to indicate the number of sufficient instances that have to complete in order for the task to terminate.

A task Q is to be executed after another task P if there is an arrow from P to Q. Tasks employ one *join* and one *split* construct. A join or split control construct may be one of the following: AND, OR, XOR, or EMPTY. Intuitively, the join specifies "how many" tasks before P are to be terminated in order to execute P, while the split construct specifies "how many" tasks following P are to be executed. The EMPTY-join (split) is used when *only one* task execution precedes (follows, respectively) the execution of P. For instance, the *Exit* task of the *BookStore* workflow in Figure 1 employs a XOR-join. Informally, *Exit* can be executed either when a token is placed into the output condition of the *GetCatalogue* task, or after the execution of the *SetOrderStatus* task if its *orderStatus* input has an "approved" value. 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. For example, the *SetPrice* task of the *Client* workflow has an OR-split and three predicates on its links to the *Payment*, *GetDeliveryInfo*, and *Exit* tasks, which decide the control-flow after its execution. Consequently, if the *price* input of *SetPrice* has a value lower than *50*, the execution of *SetPrice* is followed by the concurrent execution of *Payment* and *GetDeliveryInfo*. Otherwise, the *Exit* task is executed. Note that in the examples described in this paper we have not explicitly marked the default predicates as all predicates of the example tasks are disjoint.

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 [29], 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. (See the above discussion on executing the *Exit* task of the *BookStore* workflow.)

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. The *Client3* workflow of the example in Figure 22 employs two cancellation sets. The first one is associated to the *Wait* task and it includes the *CheckTotal*, *GetPaymentDetails*, and *RefineBookList* tasks. The second one is associated to the *GetPaymentDetails* task and it includes the *Wait* task only. The purpose of the former is to interrupt the purchase of a list of books when the *Wait* timer elapses, while the second one is used to cancel the *Wait* timer when the purchase has reached a certain task and it cannot be stopped (i.e., when the

Client3 service outputs the delivery information and the credit card details).

Please note that in this paper we use the terms *workflow* and *service* interchangeably, due to the usage of YAWL workflows to model the behaviour of (Web) services.

5 Core Aggregation

Our general aggregation methodology (introduced in [8, 10]) can be synthesised by the following phases:

1. *Service Translation*, which deals with translating real-world descriptions (e.g., BPEL + OWL ontology information, or OWL-S, etc.) of the services to be aggregated into equivalent service contracts. (In [11] we show how BPEL services can be automatically translated into YAWL workflows.)

2. *Service Matching*, which locates in a registry of service contracts candidate sets of contracts that together are able to (fully or partially) satisfy a given client contract (used as a query). Note that this phase is also in charge of (automatically) deriving a data-flow mapping among the services involved in the aggregation.

3. *Core Aggregation (and Contract Generation)*, which is applied on each candidate set obtained during the previous phase, and it deals with generating the contract of the aggregated service. Basically, this paper provides an in-depth description of this phase of the aggregation. Please note that the `Sator` methodology described here mainly enhances the core aggregation described in [8, 10] with the treatment of YAWL composite and multiple-instance tasks, as well as YAWL conditions and cancellation sets. Furthermore, this paper thoroughly illustrates the core aggregation through a few examples, as well as it a gives a first insight on our proof-of-concept prototype implementation of `Sator`.

4. *Service Deployment*, which deploys the contract of a successfully aggregated service as a real-world Web service (e.g., BPEL). (This phase is the "inverse" of the Service Translation phase.)

`Sator` (the core aggregator) 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. As previously mentioned, the service behaviour is expressed as a YAWL workflow. Atomic tasks represent simple units of work (e.g., they can be used to represent WSDL operations), and composite tasks represent complex units of work (e.g., they can be used to represent sub-services or even entire business processes).

The first step (Task Expansion) expands all tasks with explicit control- and data-flow task constructs, also called Input/Output Control/Data enabler dummies (or ICs / IDs / OCs / ODs for short). The second step (Control-Flow Analysis) translates the initial flow dependencies of each workflow in terms of the newly added IC and OC dummies. The third step (Data-Flow Analysis) relates IDs and ODs of tasks belonging to (possibly) different workflows by taking into account the data-flow mapping. The fourth and final step (Contract Optimisation) clears the aggregated contract of redundant dummies and control constructs. The four steps are detailed hereafter.

5.1 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 ontology 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.

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, while OC_A has to be connected as input of the output condition of A.

All YAWL conditions in the initial workflows, with the exception of their input and output conditions, are copied without modifications into the aggregated workflow A. Note further that the Task Expansion step works the same for atomic and composite tasks, as well as for tasks with single or multiple instances. This is because ICs and IDs represent necessary (control- and data-flow) "prerequisites" for the execution of tasks, and OCs and ODs represent

the (control- and data-flow) "effects" of executing the tasks. Such prerequisites and effects do not vary with the type of the task. Furthermore, if T is a composite task, then only T is expanded, and not the tasks it contains. This is because YAWL does not allow links to cross the boundaries of composite tasks. In other words, links cannot have the source inside a composite task and the target outside it, or vice-versa.

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

In order to ease the presentation, we shall describe in this Section the aggregation of the workflows given in Figure 1 assuming the data-flow mapping of Figure 2 from which we have removed the *GetBookPrice():price* from the second row, the *BookStore*'s column. As we shall describe in the Data-Flow Analysis step, the operations of modifying, adding, and/or removing IO matches from the data-flow mapping are a task of the client of the aggregation methodology. As previously mentioned in Section 2, for our example we assume that the *totalPrice* output of the *BookStore* service is a more adequate match for both, the *price* input of *Client* and for the *total* input of the *Bank* service, as it includes the (constant) delivery costs.

Now, the Task Expansion step applied to all the tasks of the three workflows to be aggregated yields the tasks in Figure 3. For example, the *GetCatalogue** task employs only an OD dummy as it does not have any input yet it does have an output. Furthermore, *GetBookPrice** has both an ID and an OD as it has both inputs and outputs. The join of its ID is EMPTY as *GetBookPrice* has one input only. Another example is *SetOrderStatus**, which in addition to an ID and an OD, it gets an OC, which inherits the original (XOR) split of

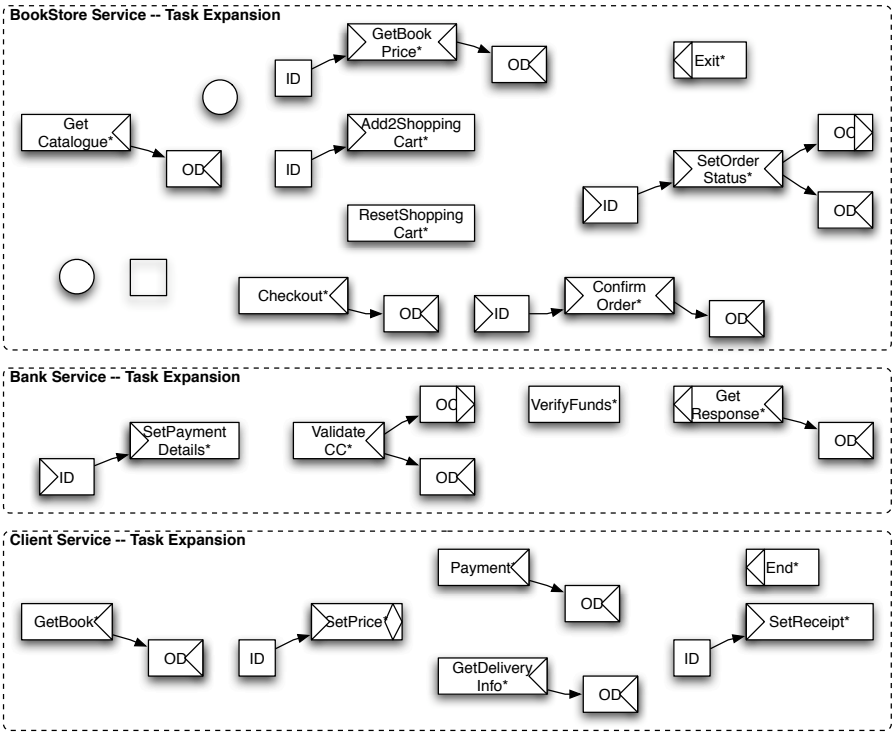

Figure 3: Task Expansion step applied to the three example workflows.

SetOrderStatus. The *Exit** tasks of both the *BookStore* and the *Client* workflows are not expanded as they do not have any inputs or outputs.

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

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

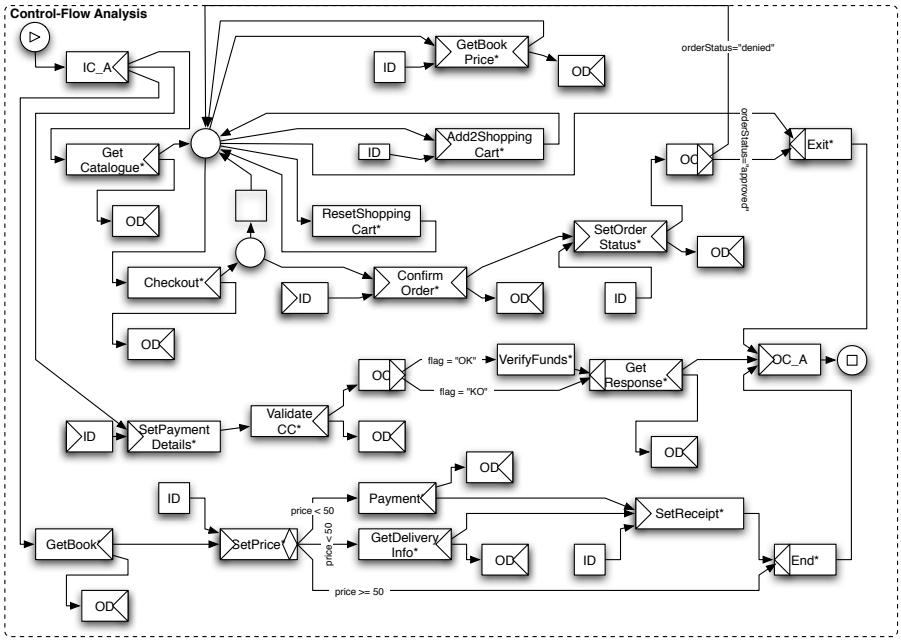

Figure 4: Applying the Control-Flow Analysis step on the example workflows.

5.2 Control-Flow Analysis

During this step, `Sator` translates the control-flow dependencies of each workflow W to be aggregated 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 (in W), `Sator` adds to A a link that points from OC_T to IC_S. Furthermore, if T was connected as input of a condition C, then `Sator` adds a link that points from OC_T to C. Analogously, if S was connected as output of a condition C, then it adds another link from C to IC_S. Note that, if T was not expanded with an OC_T dummy, then the source of the link will be T^* instead. Dually, if S was not expanded with an IC_S dummy, then the target of the link will be S^*.

By applying the Control-Flow Analysis on the three workflows of our example one gets the (partial) workflow in Figure 4.

5.3 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 (which is provided with the set of contracts to be aggregated) associates inputs and outputs of tasks belonging to (possibly) different workflows. A data-flow mapping (as the one represented by the IO matches table in Figure 2) can be simply expressed as a set of pairs $((W, T, i), (Z, S, o))$, where W and Z are two workflows, T and S are, respectively, two of their tasks, and i is an input of T, and o is an output of S. The purpose of this step is to express these mappings in terms of ID and OD dummies, as follows.

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

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

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

The Data-Flow Analysis step applied to our example translates the matches among the IOs of the services to be aggregated (see Figure 2) into dependencies among IDs and ODs of the corresponding expanded tasks. These dependencies are illustrated in Figure 5.

5.4 Contract Optimisation

The three previous steps construct the "rough" contract of the aggregated service. This last step is in charge of (repeatedly) removing from the aggregated

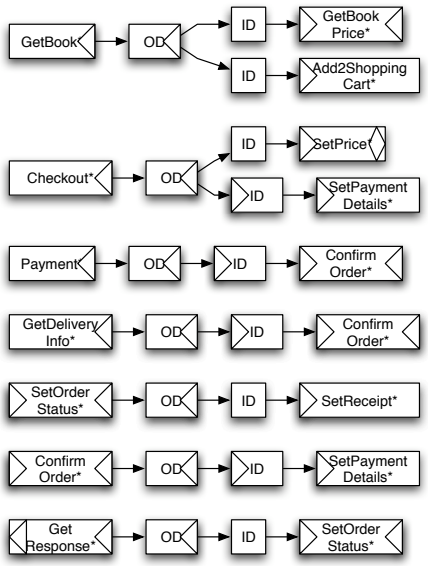

Figure 5: Expressing IO matches as dependencies among IDs and ODs.

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 contract optimisation here is not concerned with generating the "optimal aggregated workflow", which may be a topic for future work. It simply clears the "rough" workflow of redundant constructs.) 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 – $\{< EMPTY, EMPTY >, < EMPTY, \alpha >, < \alpha, \alpha >\}$ –, 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.

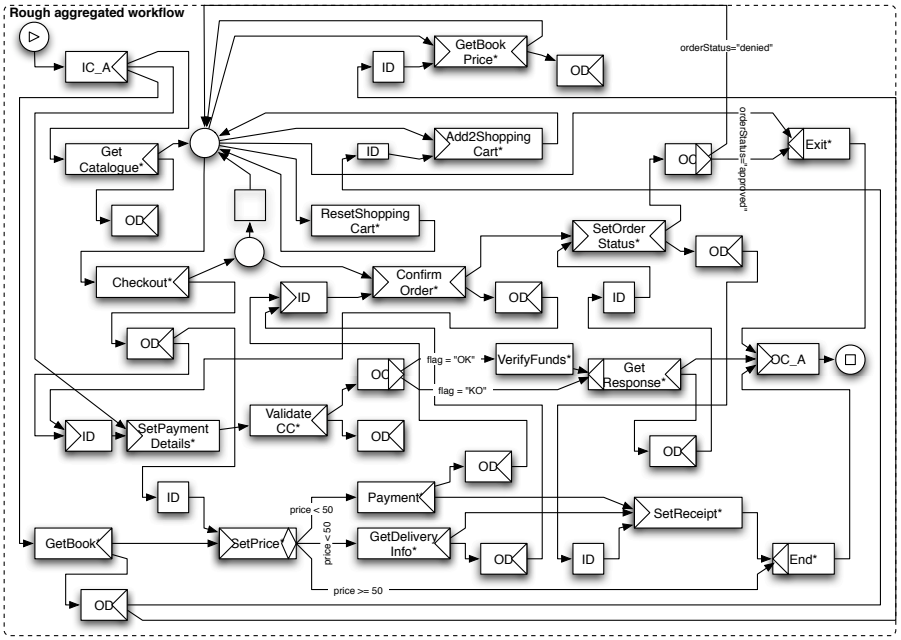

Figure 6: Rough YAWL workflow of the aggregated service.

Join/Split Elimination.

A $join_T \neq EMPTY$ has to be set to EMPTY provided T has *only 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.

The YAWL workflow of the aggregate one obtains for our example (at the end of the Data-Flow Analysis step) is given in Figure 6. We call it the "rough" workflow of the aggregate. Please note that the ODs of *ValidateCC** and *GetCatalogue** do not have outgoing links as there are no tasks whose inputs match their outputs.

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

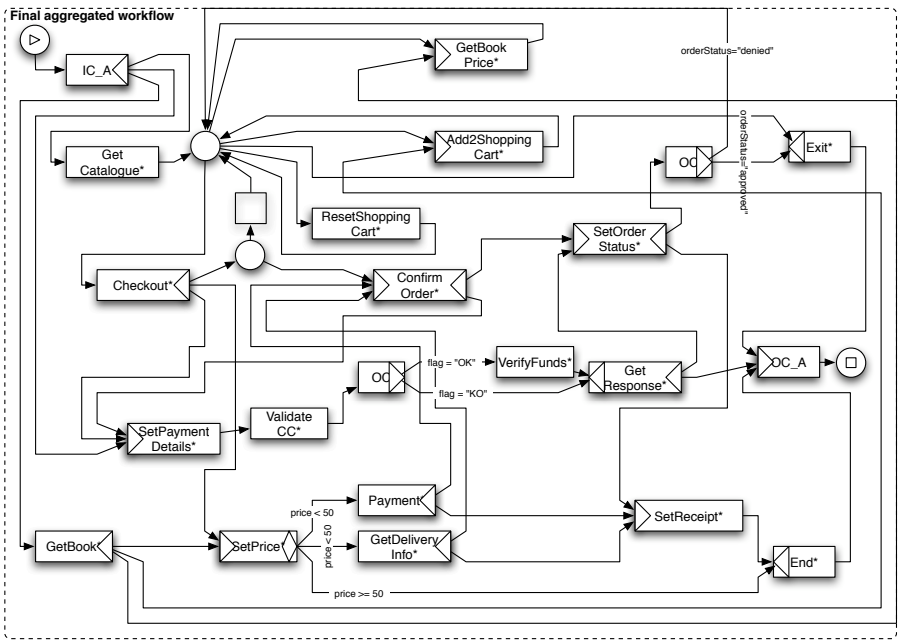

Figure 7: Final YAWL workflow of the aggregated service.

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

At the end of the Contract Optimisation step one gets the workflow in Figure 7. The only dummy which remains in the final workflow of the aggregate is the OC of the *SetOrderStatus** task. The explicit separation of the control- and the data-flow is necessary in this case as one cannot simply link the *SetOrderStatus** task as input of the output place of *GetCatalogue** task, of the *Exit** task (of the *BookStore* service), and of the *SetReceipt** task. While the execution of *SetOrderStatus** outputs a token to only one of the former two, it always sends a token to the latter. (In other words, the OC cannot be absorbed into *SetOrderStatus** as their splits are not compatible.)

5.5 Use Case: Buying a Book with the Aggregated Service

The aggregated workflow starts with the execution of the IC_A dummy, which outputs three tokens, one on each of its output links. Next, the client of the workflow may execute the *GetCatalogue** and *GetBook** tasks, which become enabled by IC_A. Note that the *SetPaymentDetails** task (corresponding to the first task of the *Bank* workflow) cannot be executed yet, as it did not receive tokens on two of its three inputs. The only token it received (from IC_A) enables it from the control-flow viewpoint. The missing tokens are to be obtained from the execution of the *Checkout* and *ConfirmOrder* tasks, which logically correspond to enabling *SetPaymentDetails** from the data-flow viewpoint (as the outputs of the former two will be used as inputs by the latter). On the one hand, the (isolated) execution of the *GetCatalogue** task enables only the *Exit**, *ResetShoppingCart**, and *Checkout** tasks, as for example, *GetBookPrice** can be executed only after the *GetBook** task (see the link between the two). On the other hand, the (isolated) execution of the *GetBook** task does not enable further tasks. However, if we assume that the client of the aggregated workflow executes *GetCatalogue** followed by *GetBook**, the following tasks become enabled {*GetBookPrice**, *Add2ShoppingCart**, *Exit**, *ResetShoppingCart**, *Checkout**}. In order to keep the use case short, suppose that the client executes first *Add2ShoppingCart** and then *Checkout**. At this point, the set of enabled tasks is {*GetBookPrice**, *Add2ShoppingCart**, *Exit**, *ResetShoppingCart**, *Checkout**, *SetPrice**}. Assume further that the client executes *SetPrice**, and that the book costs less than 50 euros. In this scenario, *SetPrice** outputs two tokens; one enables *Payment**, while the other enables *GetDeliveryInfo**. Their execution leads to enabling *ConfirmOrder**, which has now received all its needed input tokens. Note that by executing the *ConfirmOrder** task, *SetPaymentDetails** becomes the only active task. (*SetOrderStatus** is blocked waiting for a response from the *Bank* service.) Let us assume now that the client executes *SetPaymentDetails** followed by *ValidateCC**, and that the credit card she provided is valid (viz., *flag* = "OK"). The *OC* dummy of *ValidateCC** will output only one token,

which enables *VerifyFunds**. The execution of the latter can only be followed by the execution of (the "last" task of the *Bank* workflow) *GetResponse**, which unlocks *SetOrderStatus** and also sends a token to OC_A. If we suppose that the transaction was successful (viz., *orderStatus* = "approved"), the *Exit** task (which logically marks the termination of the *BookStore* workflow) outputs a token to OC_A. Next, the client can only execute *SetReceipt**, followed by (the "last" task of the *Client* workflow) *End**, which outputs the last input token needed for the execution of the OC_A task and hence for the termination of the aggregated workflow.

6 Examples

The example presented so far contains atomic processes only. In this Section we shall describe three more examples that show how `Sator` is capable of coping with:

- composite tasks,
- multiple-instance tasks, as well as with
- cancellation sets.

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

Example. The second example we shall describe in this paper is presented in Figure 8. As previously mentioned, the difference with respect to the first example introduced in Section 2 (see Figure 1) is that, here, the initial *Bank* workflow has been wrapped as a composite task. Hence, the new bank service, now called *Bank2*, "hides" its internal behaviour (viz., the content of its *ExecutePayment* task) to the other participants. By doing so one may see how the aggregation methodology copes with composite tasks.

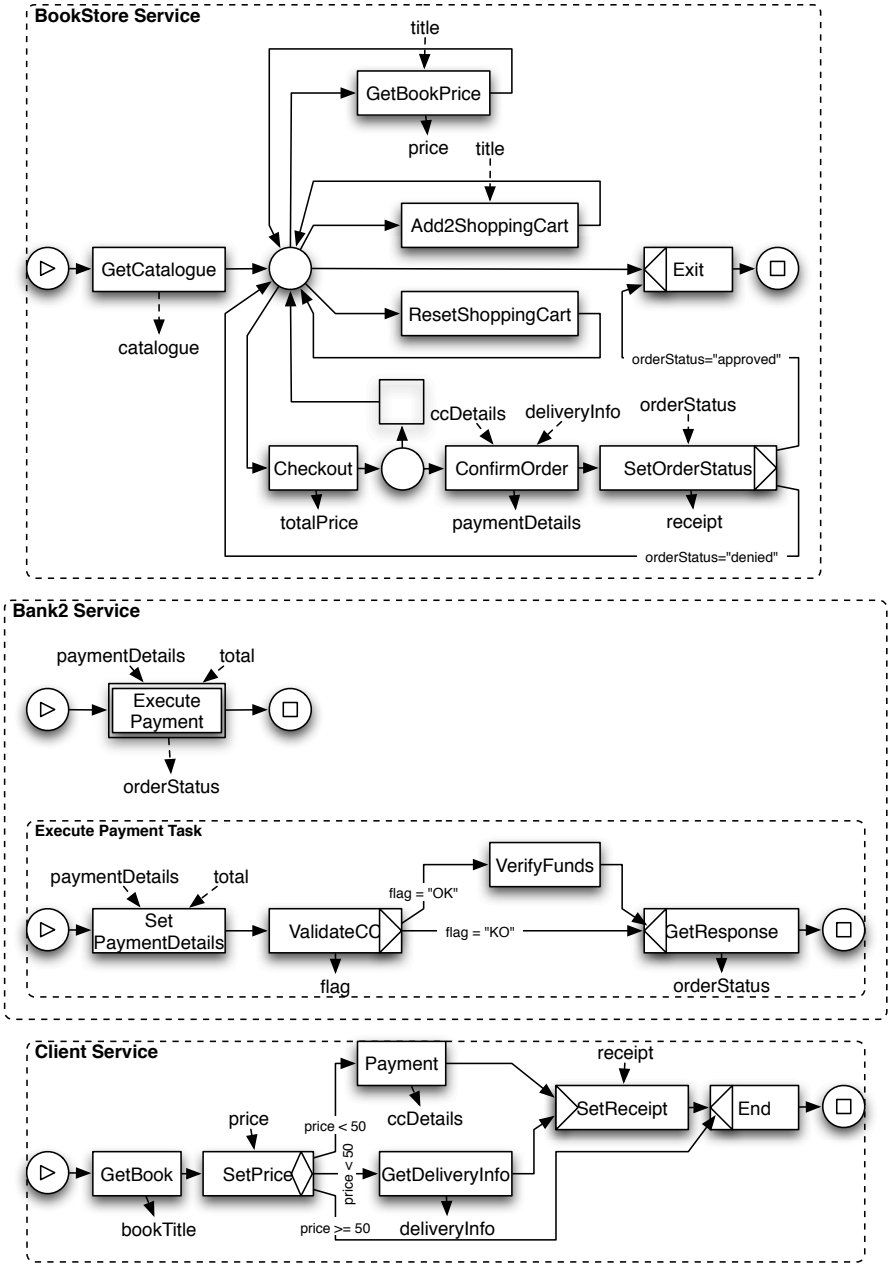

Figure 8: Another example for illustrating how the aggregation copes with composite tasks.

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

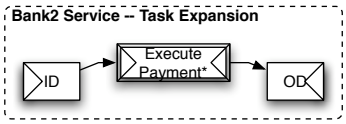

Figure 9: Task Expansion step applied to the *Bank2* service.

Task Expansion.

We recall that the Task Expansion step serves to explicitly split the control- from the data-flow dependencies. The Task Expansion step applied to the *Bank2* workflow gives the three tasks in Figure 9. We recall that only the tasks of the top-level workflow net [29] can be expanded. In other words, the Task Expansion step cannot be applied to the workflow net of the *ExecutePayment* composite task as links that might be added during the Data-Flow Analysis cannot cross the boundary of the composite task. Note in Figure 9 the AND-join of the ID, which is due to the fact that *ExecutePayment* has two inputs.

Control-Flow Analysis.

The Control-Flow Analysis step translates the initial control-flow dependencies of each workflow to be aggregated into dependencies among control-flow dummies of the expanded tasks. A task should be used instead of the dummies if the respective task was not expanded with control-flow dummies. The Control-Flow Analysis yields the (partial) workflow given in Figure 10. Note the simplification of the workflow due to the encapsulation of the *Bank* service logic as a composite task. The initial control-flow links between the *ExecutePayment* task and the input and output conditions of the *Bank2* workflow are translated into links between the expanded *ExecutePayment** task and the IC_A and the OC_A, respectively, dummies of the aggregate.

Data-Flow Analysis.

The new matches among IOs of the three services in Figure 8 are presented in Figure 11. Observe that the *Bank2* column refers only to the *ExecutePayment*

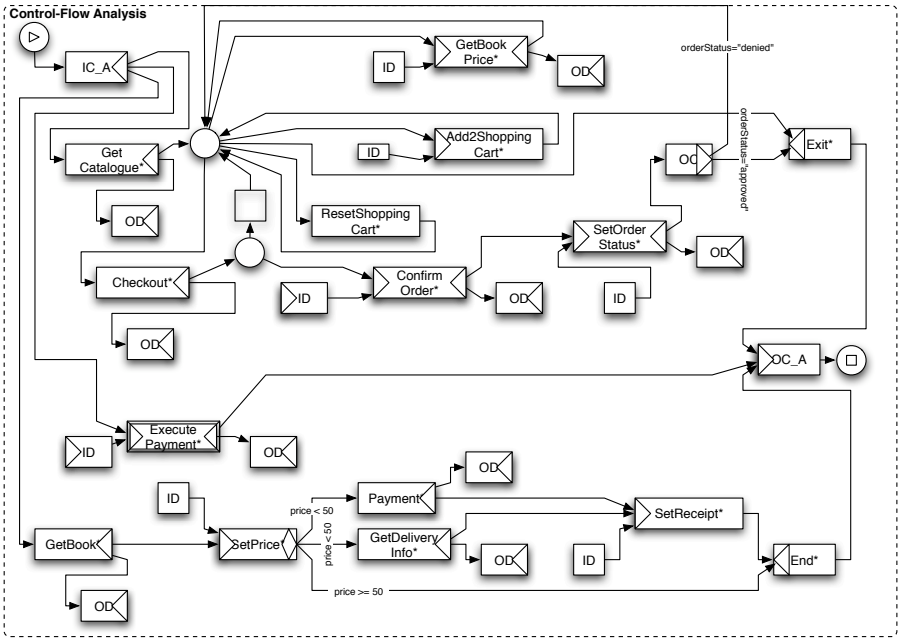

Figure 10: Control-Flow Analysis of the example workflows in Figure 8.

task. Furthermore, the last row of the IO matches table of the first example (see Figure 2) is not included in the new table, as the *ValidateCC* task outputting the *flag* parameter is now "hidden" inside the *ExecutePayment* task.

Consequently, the Data-Flow Analysis now links the ODs of *Checkout** and *ConfirmOrder** with the ID of *ExecutePayment**, and the OD of *ExecutePayment** with the ID of *SetOrderStatus**. The transformation of the IO matches into workflow dependencies linking IDs and ODs is given in Figure 12.

Furthermore, the "rough" workflow of the new aggregated service, which one obtains at the end of this step, is given in Figure 13.

Contract Optimisation.

This step is quite similar to the previous example. The Dummy Absorption and Join/Split Elimination criteria remove all the redundant IDs and ODs and reset to EMPTY the joins and splits with one input and output, respectively. Figure 14 shows the final YAWL workflow of the aggregate. Note that in order

Client	BookStore	Bank2
GetBook():bookTitle	GetBookPrice(title):... --- Add2ShoppingCart(title):...	-
SetPrice(price)	GetBookPrice():price --- Checkout():totalPrice	ExecutePayment(total):...
Payment():ccDetails	ConfirmOrder(ccDetails):...	-
GetDeliveryInfo():deliveryInfo	ConfirmOrder(deliveryInfo):...	-
-	GetCatalogue():catalogue	-
-	ConfirmOrder():paymentDetails	ExecutePayment(paymentDetails):...
-	SetOrderStatus(orderStatus):...	ExecutePayment():orderStatus

Figure 11: The new IO matches table among parameters of the three services in Figure 8.

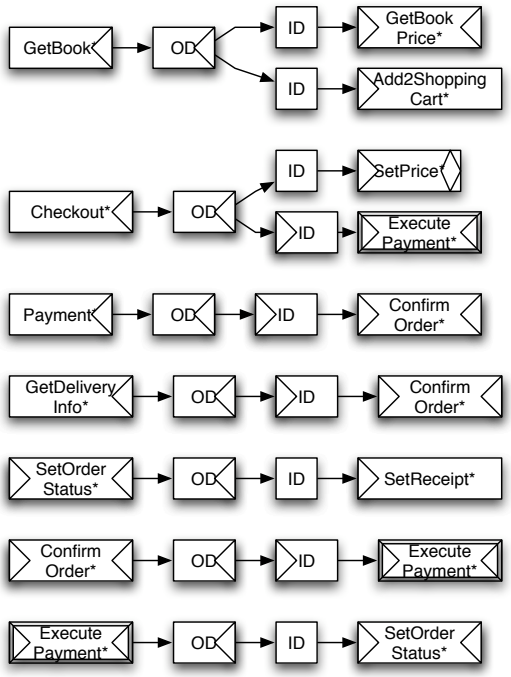

Figure 12: Transformation of the IO matches into dependencies among IDs and ODs.

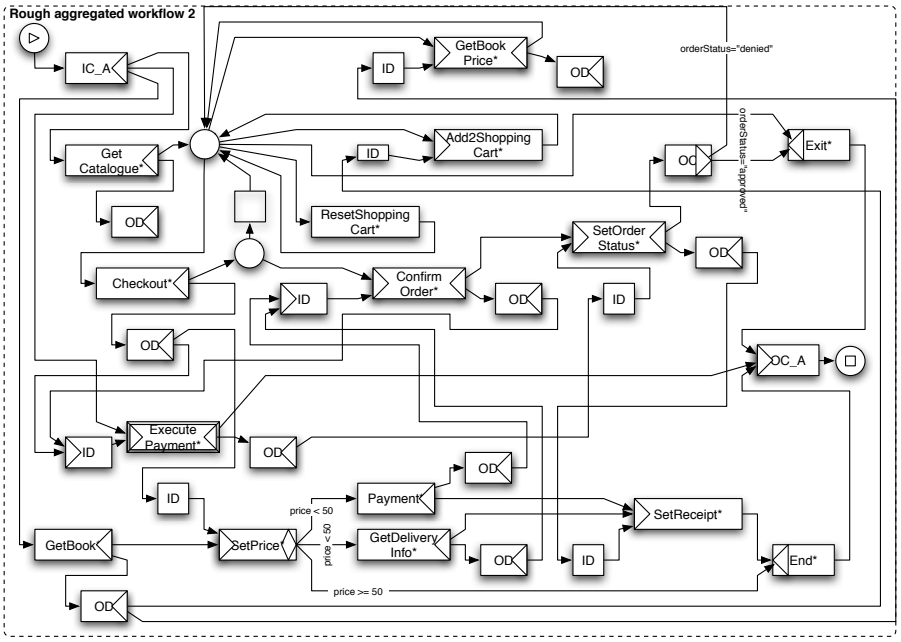

Figure 13: The rough workflow of the service obtained by aggregating the three services in Figure 8.

to execute the *ExecutePayment** task one has to enable it first, both from the control- and the data-flow viewpoints. The former relates to executing the IC_A task, while the latter to executing the *Checkout** and the *ConfirmOrder** tasks.

The scenario for buying a book with this aggregated service is quite similar to the one of the first example (see Subsection 5.5), and hence we shall not describe it here. (However, note that in this case the execution of the *ExecutePayment** composite task leads to the execution of the tasks contained in its workflow net.) ◇

Example. The third example is introduced in Figure 15. As previously mentioned, in this example we modify the *Client* and the *BookStore* services. They are called now *Client2* and *BookStore2*, respectively.

The *Client2* workflow starts with the execution of the *ChooseBooks* task, which inputs a *catalogue* of books and it outputs a list of books to be bought. Next, the *CheckTotal* task waits for the user to input a maximum price to be

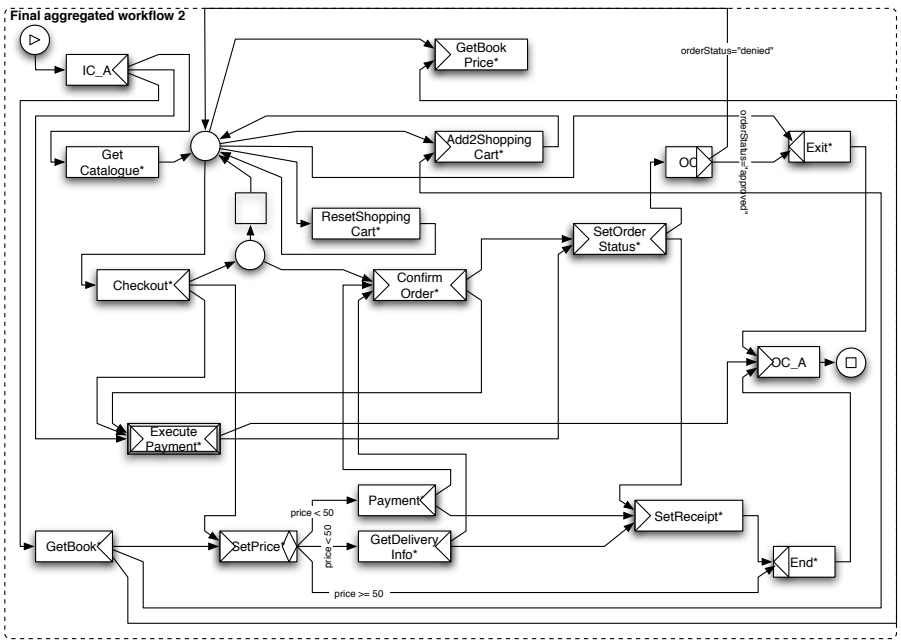

Figure 14: The final workflow of the service obtained by aggregating the three services in Figure 8.

paid for these books (*maxPrice*), as well as it waits for a list of book prices from the *BookStore2* service. The *booksPrice* output of *CheckTotal* stands for the total cost of the books (excluding delivery costs). Then, the control-flow is decided based on the *booksPrice* and on the *maxPrice*. On the one hand, if all books can be bought, *Client2* first executes *GetPaymentDetails*, which outputs the delivery information and the card details, and then it executes the *SetReceipt* task, which inputs the *receipt* from the *BookStore2* service. On the other hand, if *booksPrice* exceeds *maxPrice*, the execution of the workflow continues with a deferred choice. The invoker of the *Client2* service has to decide whether to exit by executing the *End* task, or to remove some of the books from the *selectedBooks* list. *RefineBookList* inputs *priceList* so as to ease the job of the invoker by displaying the price of each book in the list. The execution continues next with the *CheckTotal* task.

The main difference between the (new) *BookStore2* and the (old) *BookStore* workflows, is that *BookStore2* has a multiple-instance task – *GetBookPrices*,

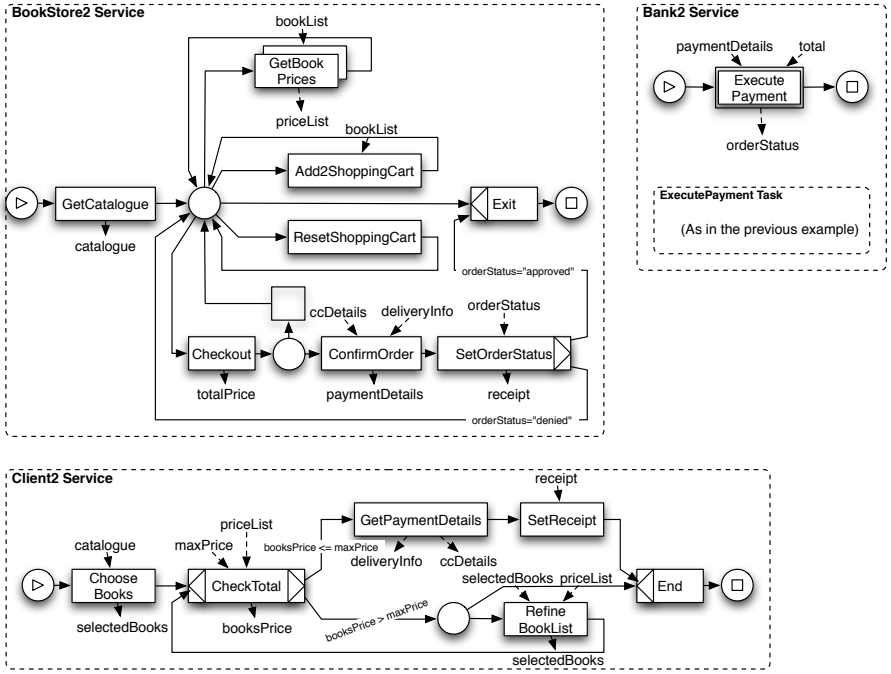

Figure 15: Example for illustrating how the aggregation copes with multiple-instance tasks.

which inputs a list of books and it outputs a list containing their prices. We assume that the number of instances of the *GetBookPrices* task is fixed and equal to the size of the *bookList*, same as the lower and the upper bounds of the number of instances created after the initiation of the task, and the threshold value that decides when the *GetBookPrices* task completes its execution. (For more information on multiple-instance tasks please see [29].) Hence, each book in the *bookList* leads to an instance of the *GetBookPrices* task, which outputs the book's price. When all instances have finished their executions, the output of *GetBookPrices* is obtained by merging the individual book prices into the *priceList*. This behaviour is achieved by suitably mapping the IOs of the *GetBookPrices* task and of the workflow net of the *BookStore2* service, yet going into such technical depths it is out of the scope of this paper.

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

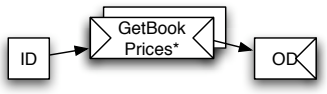

Figure 16: Expanding the *GetBookPrices* task of the *BookStore2* workflow.

TaskExpansion.

The Task Expansion step expands the tasks of the three workflows as shown in the previous examples. Consequently, we shall present here only the expansion of the multiple-instance task *GetBookPrices* of the *BookStore2* workflow. As illustrated in Figure 16, *GetBookPrices** employs AND-join and -split constructs, as well as it is connected with an ID and an OD task.

Informally, the ID enables *GetBookPrice** from the data-flow point-of-view, that is, it waits for a value to be mapped to the *bookList* input parameter of *GetBookPrice**, while the OD broadcasts its *priceList* output. Hence, from the Task Expansion viewpoint, a multiple-instance task (similarly to a composite task) looks exactly like a simple atomic task.

Control-Flow Analysis.

This step builds (part of) the control-flow of the aggregate by translating the initial control-flow links among workflow tasks into links among ICs and OCs. Applying this step to the third example in Figure 15 yields the partial workflow in Figure 17.

Data-Flow Analysis.

Matching the IO parameters of the workflows in this example leads to the table in Figure 18. One may see that the resulting table is slightly more complicated with respect to the previous examples due to the increased number of matches. For example, the fifth row describes the fact that the *selectedBooks* input of *RefineBookList* matches similar outputs of *ChooseBooks* and *RefineBookList* of the same workflow (*Client2*), as well as the *bookList* outputs of the *GetBookPrices*

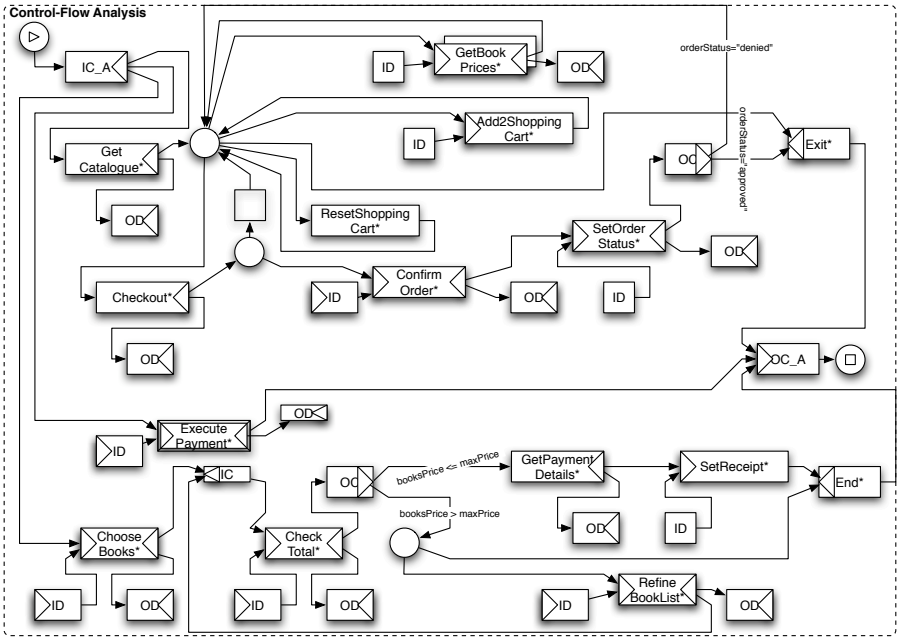

Figure 17: Control-Flow Analysis of the example workflows in Figure 15.

and *Add2ShoppingCart* tasks of the *BookStore2* workflow. In this example we shall assume that the client of the aggregation process removes only the matches between the *maxPrice* input and *booksPrice* output of the *CheckTotal* task (of the *Client2* workflow) with the *totalPrice* output of the *Checkout* task (of the *BookStore2* workflow), and with the *total* input of the *ExecutePayment* task (of the *BankService2* workflow). In other words, the second row, first column of the table in Figure 18 is set to void. On the one hand, the removal of *maxPrice* is (mainly) motivated by the fact that it is an input of the *Client2* service, whose value has to be provided by the invoker of the *Client2* service, and not taken from the output of another service in the aggregation. On the other hand, the removal of *booksPrice* is due to the fact that it is an internal flag-variable used to decide the control-flow following the *CheckTotal* task.

The ontology matches in Figure 18 (after removing the unwanted matches) relate in terms of dependencies among ID and OD dummies as shown in Figure 19. Please note the dummies necessary when an input matches sev-

Client2	BookStore2	Bank2
ChooseBooks(catalogue):...	GetCatalogue():catalogue	-
CheckTotal(maxPrice):... --- CheckTotal():booksPrice	Checkout():totalPrice	ExecutePayment(total):...
CheckTotal(priceList):... --- RefineBookList(priceList):...	GetBookPrices():priceList	-
SetReceipt(receipt)	SetOrderStatus():receipt	-
RefineBookList(selectedBooks):... --- ChooseBooks():selectedBooks --- RefineBookList():selectedBooks	GetBookPrices(bookList):... --- Add2ShoppingCart(bookList):...	-
GetPaymentDetails():ccDetails	ConfirmOrder(ccDetails):...	-
GetPaymentDetails():deliveryInfo	ConfirmOrder(deliveryInfo):...	-
-	ConfirmOrder():paymentDetails	ExecutePayment(paymentDetails):...
-	SetOrderStatus(orderStatus):...	ExecutePayment():orderStatus

Figure 18: The IO matches table among parameters of the three services in Figure 15.

eral outputs. This is the case for the *bookList* inputs of *GetBookPrices* and *Add2ShoppingCart*, as well as for the *selectedBooks* input of *RefineBookList*. Each such input dummy has a XOR-join as one (output) value only is enough for mapping the respective (input) parameter. For example, a *bookList* input for *GetBookPrices* can be obtained *either* from the output of *ChooseBooks*, *or* from the output of *RefineBookList*. It is sometimes the case that some of the data dependencies are redundant. This is the case of the (data-flow) loop created around *RefineBookList** due to the match between its *selectedBooks* input and its *selectedBooks* output. Usually, avoiding the generation of such loops is the task of the aggregation client. She can either check the data-flow mapping (viz., the IO matches table) "by hand", or she can use tools implementing the method-

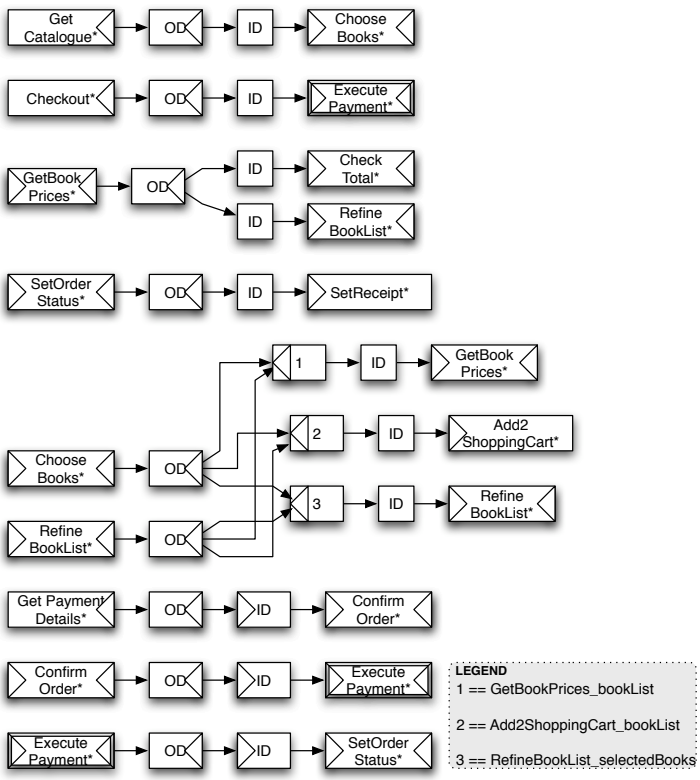

Figure 19: Transformation of the IO matches into dependencies among IDs and ODs.

ology described in [8] e.g., for the detection of (dead-)locks in the aggregated workflow. Should a (dead-)lock exist, she can (manually) remove the troublesome match(es) from the data-flow mapping, and then redo the (automated) core aggregation process.

The rough aggregated workflow reflecting both control- and data-flow dependencies among the participant services is given in Figure 20. Please note that the dummy task joining in input the ODs of *ChooseBooks** and *RefineBookList** has not been produced by the aggregation methodology. We use it here just for simplifying a bit the graphical representation of the control-flow of the aggregated service.

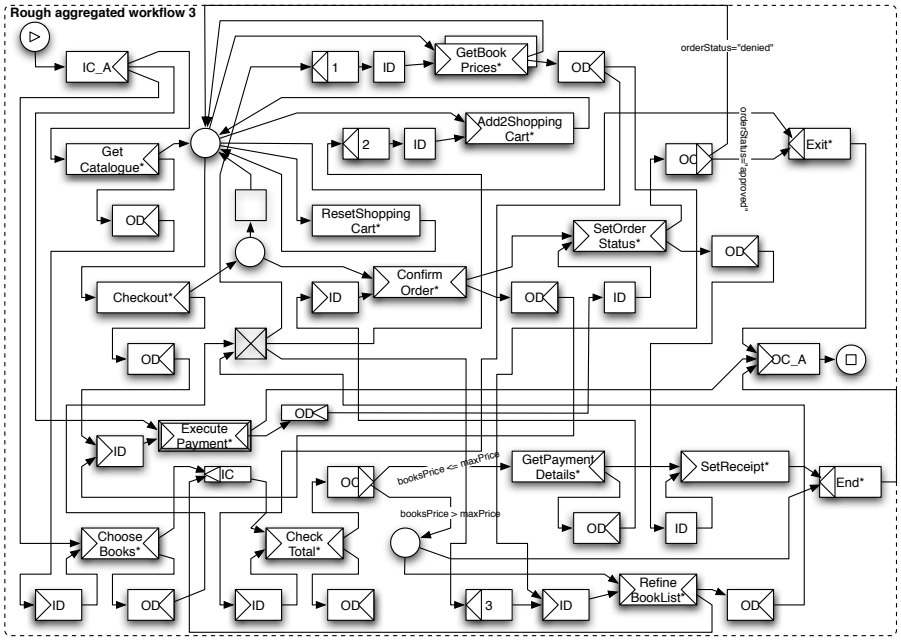

Figure 20: The rough workflow of the service obtained by aggregating the three services in Figure 15.

Contract Optimisation.

After removing redundant dummies as well as redundant joins and splits from the rough workflow of the aggregate, one obtains the workflow in Figure 21. Note that the aggregation removes the IDs of *GetBookPrices**, *Add2ShoppingCart**, and *RefineBookList**, yet not their input dummies added during the Data-Flow Analysis phase (i.e., *GetBookPrices_bookList*, *Add2ShoppingCart_bookList*, and *RefineBookList_selectedBooks* respectively, denoted by *1*, *2*, and *3* in Figure 21). It is interesting to note that the *CheckTotal** in Figure 21 is obtained by:

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

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

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

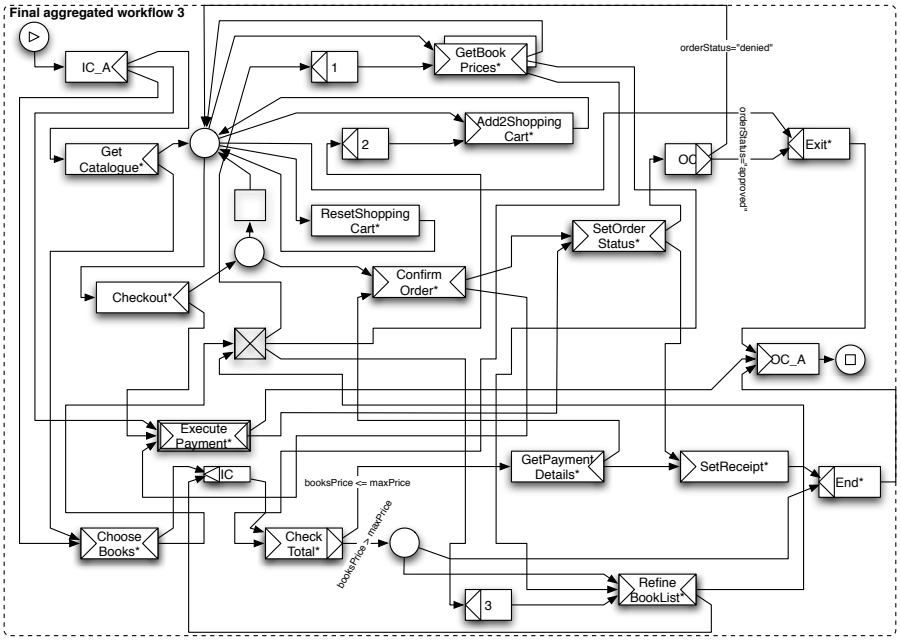

Figure 21: The final workflow of the service obtained by aggregating the three services in Figure 15.

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

Example. For our last example, we shall add a cancellation set to the *Client2* workflow, which is in charge of cancelling the purchase of a list of books at a certain timeout. The workflows to be aggregated are given in Figure 22.

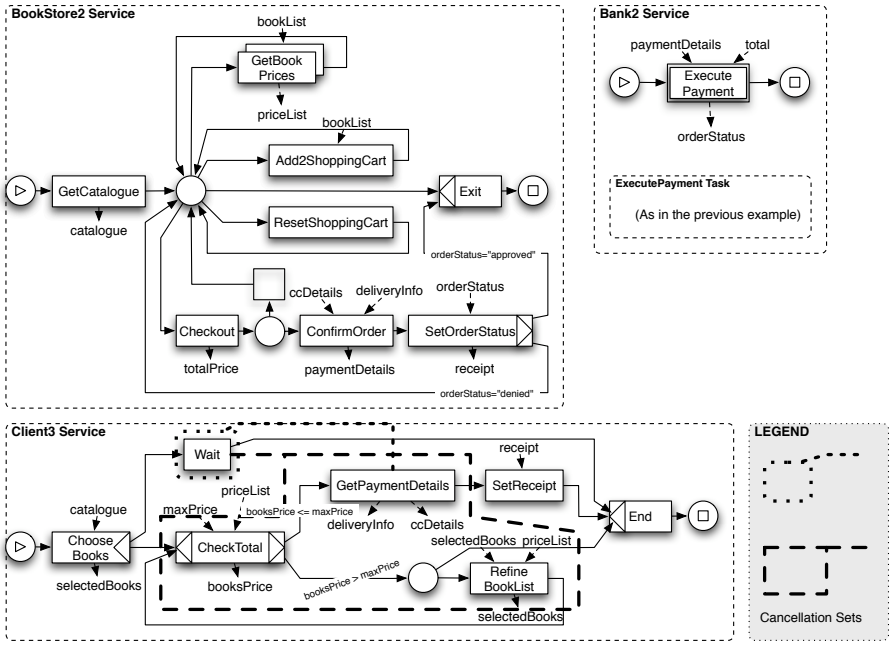

Figure 22: Final example for illustrating how the aggregation copes with cancellation sets.

The *Client* workflow, now called *Client3*, starts with the execution of the *ChooseBooks* task, as in the previous example. However, after executing *ChooseBooks*, the workflow executes concurrently the *CheckTotal* and the *Wait* tasks. Basically, the execution of the *Wait* task resumes to waiting for a certain amount of time t, which is given as input. (Please note that we have not represented the input of *Wait*, as well as we shall not go into any details about the YAWL *TimeService* implementing the *Wait* task as they are not crucial for the presentation of the aggregation methodology.) When the amount of time t has elapsed (viz., the *Wait* task has finished its execution), the YAWL engine removes all tokens from the cancellation set of *Wait*. Hence, the *Wait* task is in charge of cancelling the purchase of a list of books given a time period has elapsed. In this scenario, the execution of the workflow finishes as *Wait* outputs a token for the *End* task. The second cancellation set associated to the *GetPaymentDetails* task serves to cancel the *Wait* timer. The execution of the *GetPaymentDetails* task invalidates the execution of the *Wait* task in order to prevent the cancel-

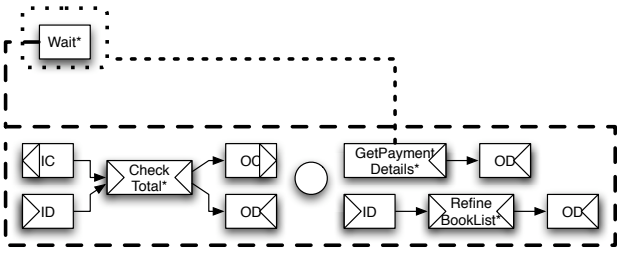

Figure 23: Expanding tasks included in, or associated to cancellation sets.

lation of the purchase when the *Client3* workflow has outputted the credit card details and the delivery address.

TaskExpansion.

The particularity of this example is the usage of cancellation sets. As described in Section 5, if a task X is belongs to a cancellation set, then the Task Expansion step basically includes in the respective cancellation set all expansion dummies of X. For example, Figure 23 illustrates the expansion of the four tasks of the *Client3* workflow belonging to the two cancellation sets. On the one hand, the cancellation set of *Wait** includes the IC/IDs and OC/ODs of the three other tasks, while the cancellation set of *GetPaymentDetails** includes *Wait** only. Furthermore, the condition in the cancellation set of *Wait* is included into the cancellation set of *Wait** as well.

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

The Control-Flow Analysis step does not change when dealing with cancellation sets. Consequently, the rough aggregate for this example is quite similar to the one obtained for the previous example (see Figure 20). The main add-on of this rough aggregated workflow consists of the two cancellation sets, as described in the Task Expansion step (see Figure 23). This is mainly due to the fact that the only new task of this example is *Wait*, which adds nothing to the previously

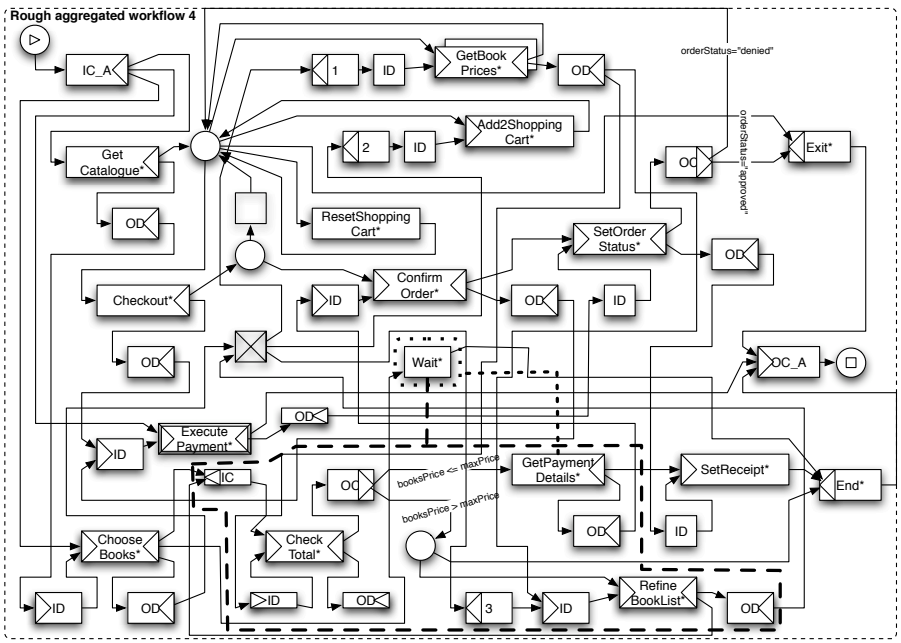

Figure 24: The rough workflow of the service obtained by aggregating the three services in Figure 22.

obtained IO matches table[4] (see Figure 18).

As explained in the previous example, the Data-Flow Analysis adds three dummies for dealing with multiple output matches for the *bookList* inputs of *GetBookPrices* and *Add2ShoppingCart*, and for the *selectedBooks* input of *RefineBookList*. While the former two dummies do not lead to any changes in the aggregated workflow, it is important to note that the *RefineBookList_selectedBooks* dummy has to be added to the cancellation set of $Wait^*$.

Also with respect to cancellation sets, the Contract Optimisation step acts by removing dummies from cancellation sets when they are absorbed into other tasks. For example, this is the case of the ID and OD dummies of the $CheckTotal^*$ task (see Figure 24).

The final aggregate workflow of this example is depicted in Figure 25. Note that after removing all redundant dummies, the cancellation set of *Wait* still

[4]Please see the discussion at the beginning of this example, in which we motivate why we do not represent the input of the *Wait* task.

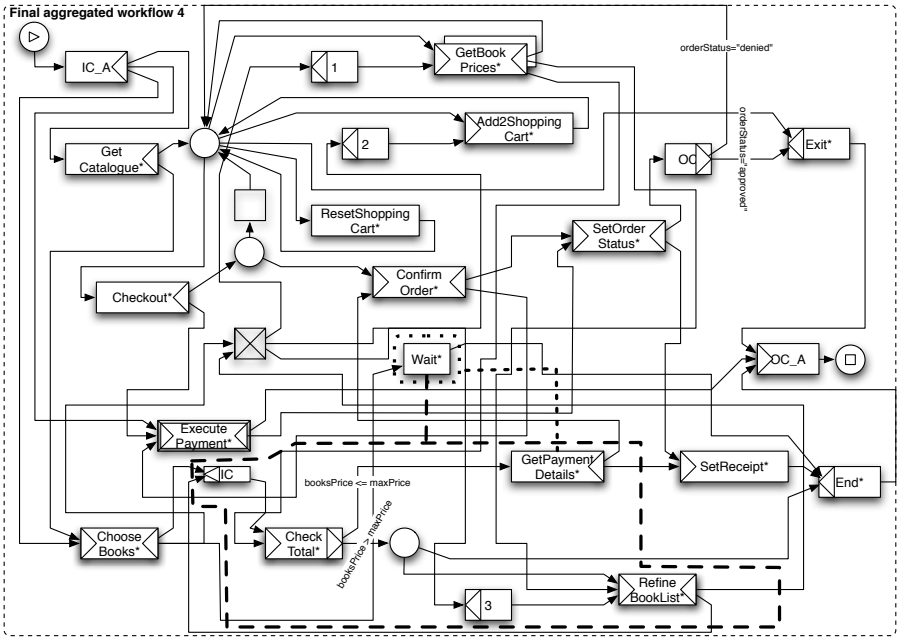

Figure 25: The final workflow of the service obtained by aggregating the three services in Figure 22.

includes the *RefineBookList_selectedBooks* (denoted by 3 in Figure 25) input dummy of *RedefineBookList**.

As one may have noted, there are two possible execution scenarios for this aggregated workflow. On the one hand, if the purchase of the books ends before the *Wait** timer elapses, the execution behaviour of the aggregate is quite similar to the previous example. The main difference is that here the execution of the *GetPaymentDetails** task leads to the removal of all the tokens in its cancellation set, and consequently to the cancellation of the timer. On the other hand, if *Wait** terminates (viz., the timer elapses) before *GetPaymentDetails** does, then the entire aggregated workflow locks as, for example, the *ConfirmOrder** task blocks waiting for the payment details. It is important to note that the lock is due to a behavioural mismatch between the participant workflows, and not due to a flaw in the aggregation process. (In [8] we show how a reachability analysis of YAWL workflows can be employed to verify e.g., lock-freedom, while in [7] we present an adaptation methodology for tackling behavioural mismatches.) $\diamond$

The interested reader can download the examples described in this paper from `http://www.di.unipi.it/~popescu/Sator_Examples.zip`.

7 Implementation

In this section we discuss the main implementation aspects (e.g., choice of data structures, marshalling and unmarshalling of YAWL workflows, etc.) of our Java proof-of-concept prototype implementation of `Sator`, which was previously described in Section 5. Furthermore, we include some words on the Java packages implementing the aggregation as well as a URL for downloading the source code of *Sator*.

Implementation Choices.

The main implementation choices were conditioned by the following aspects:

- Selection of the programming language for the implementation,
- Transposition of YAWL workflows from a XML representation into data structures, on which the aggregation methodology can be applied,
- Format and acquisition of the data-flow mapping (i.e., a set of dependencies among inputs and outputs of tasks belonging to different workflows), and
- Deployment of the data structures produced by the aggregation process into a XML file representing the aggregated workflow.

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

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

- *Forward compatibility* (with respect to the YAWL engine and editor): the YAWL deployment files are tied up through a XML Schema and, whenever new versions of the YAWL tools are released, this schema can be updated.

With respect to the data structures, we preferred to adopt those defined in the YAWL code library, as they are both the result of the unmarshalling process and the needed starting point for the marshalling phase. However, as future work, we plan to introduce an intermediate step to convert YAWL data structures into a set of data structures specifically optimised for the aggregation algorithm, thus making the implementation more efficient in aggregating large sets of services.

As for the format of the data-flow mapping, we chose a simple XML format, for homogeneity reasons with the rest of the input files. In [8, 9] we showed how ontology-based matching can be applied to automatically derive the data-flow dependencies linking workflow tasks from the semantic descriptions of the services to be aggregated.

Main Implementation Solutions.

The main implementation solutions can be synthesised as follows:

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

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

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

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

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

Code Structure and Code Quality Evaluation.

The implementation consists of three packages: $wsa.aggregation$, $wsa.support$, and $wsa.user\_interface$. The first one contains the $AggregatedYSpecification$ class, which holds the aggregated workflow and the methods relative to the aggregation methodology phases. The $wsa.support$ package contains some record classes used to pass complex data during the aggregation, and some support methods used to work out some low-level problems such as transposition of

mappings between global variables and process parameters of the starting services, production of unique identifications for global variables in the aggregated workflow and so on. Finally, the $wsa.user\_interface$ includes the classes concerning the GUI of the aggregator.

Furthermore, the source code of `Sator` is freely usable, modifiable and redistributable under GPL license. The interested reader can download it from `http://www.di.unipi.it/~popescu/Sator_SourceCode.zip`.

8 Related Work

In this section we briefly discuss other manual, semiautomatic, and 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) [5]. 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. [37] define the Service Scheduling Language and the Service Composition Execution language, and manually produce sequential or concurrent service compositions from simple or complex Web services wrapped as components.

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. [13]. 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. [17] model the control-flow of the desired composed service while service advertisements are described through their IOs only. The composition is done by matching requested operations with the advertised ones based on IOs and non-functional properties.

The **automatic** composition of services 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. [25] 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. proposed SWORD [22] 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. [16] adapted Golog, (a high-level logic programming language based on situation calculus), for the composition of Semantic Web services (McIlraith, 2002). 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 [36] 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. [26] 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. [4] 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 is that it is difficult to represent the goal. Furthermore, A.I. approaches are computationally expensive.

Several reviews accurately describe current trends in Web services composition. In [12], 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 [14], while another one between BPEL and WSCI is given in [38]. 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 [31].

Preliminary versions of the *Sator* core aggregation methodology described in this paper have been presented in [8, 10]. The present paper extends [8, 10] by tackling YAWL conditions, composite tasks and multiple task instances, as well as cancellation sets. Furthermore, in this paper we thoroughly illustrate the core aggregation through a few examples and we give a first insight on our proof-of-concept Java prototype implementation of `Sator`. 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:

- It is amenable to efficient implementations, as it relies on service contracts, which can be generated off-line,

- It can be employed to discover [8], aggregate [8, 10], and adapt [7, 9]

BPEL processes, as it straightforwardly integrates with the BPEL2YAWL translator described in [11], as well as

- It provides the basis to discover, aggregate, and adapt 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.

9 Conclusions

In this paper we have presented `Sator`, the core of a (Web) service aggregation methodology that, given a set of advertised service contracts together with a data-flow mapping linking service parameters, automatically generates the contract of a composite service. The long-term goal of our aggregation methodology is to compose services written with different service description languages such as BPEL [5] or OWL-S [19]. A key ingredient of our framework is the notion of service contract consisting of an ontology-annotated signature and of a behaviour 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 aggregrating 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 [15], and ontology-aware matching algorithms such as [6, 8, 20] can be exploited 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 (e.g., so as to cope with cross-ontology mapping). We chose YAWL [29] 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.

Following [18], 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. `Sator` generates the workflow of the composite from the initial workflows by suitably adding control-flow constraints among their tasks due to data-flow dependencies among parameters. 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.

Future work will mainly be devoted to the integration of `Sator` into the extended aggregation methodology described in [8], as well as to applying the adaptation methodologies described in [7, 9] 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] 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.

[5] BPEL4WS Coalition. Business Process Execution Language for Web Services (BPEL4WS) Version 1.1.

(ftp://www6.software.ibm.com/software/developer/library/ws-bpel.pdf).

[6] 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.

[7] A. Brogi and R. Popescu. Automated Generation of BPEL Adapters. Technical Report, University of Pisa, 2006. (http://www.di.unipi.it/~popescu/BPELAdapters.pdf).

[8] A. Brogi and R. Popescu. Contract-based Service Aggregation. Technical Report TR-06-12, University of Pisa, April 2006. (http://compass2.di.unipi.it/TR/Files/TR-06-12.pdf.gz).

[9] A. Brogi and R. Popescu. Service Adaptation through Trace Inspection. In S. Gagnon, H. Ludwig, M. Pistore, and W. Sadiq, editors, *Proceedings of SOBPI'05*, pages 44–58, 2005. (http://elab.njit.edu/sobpi/sobpi05-proceedings.pdf).

[10] A. Brogi and R. Popescu. Towards Semi-automated Workflow-Based Aggregation of Web Services. In B. Benatallah, F. Casati, and P. Traverso, editors, *ICSOC'05*, volume 3826 of *LNCS*, pages 214–227. Springer, 2005.

[11] A. Brogi and R. Popescu. From BPEL Processes to YAWL Workflows. In M. Bravetti, M. Nuňes, and G. Zavattaro, editors, *Proceedings of the 3rd International Workshop on Web Services and Formal Methods WS-FM 2006, Vienna, Austria, September 8-9 2006*, volume 4184 of *LNCS*, pages 107–122. Springer, 2006.

[12] 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.

[13] 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.

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

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

[16] 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.

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

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

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

[20] 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.

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

[22] 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.

[23] 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.

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

[25] 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.

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

[27] UDDI Coalition. The UDDI Technical White Paper. (`http://www.uddi.org/`).

[28] 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`).

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

[30] 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.

[31] 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.

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

[33] WSCI Coalition. Web Service Choreography Interface (WSCI) 1.0. (http://www.w3.org/TR/wsci).

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

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

[36] 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.

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

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