INFORMATYKA EKONOMICZNA BUSINESS INFORMATICS 3(29) • 2013
ISSN 1507-3858

Lev Belava
AGH University of Science and Technology, Krakow
e-mail: Lev.Belava@gmail.com

TOWARDS A PLATFORM FOR HYBRID COMPOSITION AND GROUNDING OF WEB SERVICES

Abstract: This paper describes an approach to a software platform and a method of hybrid web services composition and hybrid grounding of abstract composition plans. It also presents in a highly detailed manner the architecture of the implemented platform and its modules.

Keywords: SOA, web services composition, web services grounding.

1. Introduction

There is a clear trend in modern science and business communities to switch from old monolithic style software platforms towards architectures that favor changing and switching their component modules. Generally speaking, modularity benefits all parties that are engaged in the development, usage and maintenance of properly constructed modular software systems. Business users can receive new functionalities or change the existing ones much easier and quicker. The maintenance process gets more refined since usually it is an easier task to fix and support a number of relatively simple, small and well-defined modules than to take care of big monolithic all-in-one systems which have tendencies to accumulate a number of unknown internal code and logic interdependencies and other pitfalls. The programming of clear and well-defined software components is an easier task, too. Such modules have to be less dependent on other platform components and are smaller since they need to implement only a portion of system functionality.

A Service Oriented Architecture approach and its most common implementation – Web Services – naturally fit this general trend and push it even further: software modules have now become standalone applications in a network environment. They are called services and, by working together, can offer various complex functionalities for their users. The practice of SOA service composition is also an interesting and promising concept of developing new software systems with a highly refined functionality that is achieved by using combinations of different services.

There are two main parts of such software systems – service composition and grounding. Service composition focuses mainly on methods and algorithms that can produce viable composition plans from sources like sets of available web services or other plans. On the other hand, grounding focuses on transforming abstract composition plans into execution ready plans. During the grounding process every abstract service from the abstract composition plan has to be associated with a real-world service instance so that it can be used in the service execution process.

1.1. Service Composition and Grounding

SOA is a very general software architecture and engineering paradigm. It does not define services very strictly, moreover, it does not even specify that services should exist in a network. It only presumes that services have to be relatively independent from each other and offer functionalities formally described by their interface. Web Services are the most common SOA implementation nowadays.

Service composition is a concept of combining different services for data processing purposes. It enables users to create complex processes by combining various functionalities offered by available services. To date, numerous service composition techniques have been developed.

Manual and semi-automatic service composition methods e.g. [Sirin, Hendler, Parsia 2003] and [Sirin, Hendler, Parsia 2004] are relatively popular in the scientific community. Such kinds of approaches are fairly easy to understand and implement. When creating service compositions, all decisions are made by the user who is provided with some kind of advice or narrowing choice options at the most. However, such methods do not offer service composition process automation.

Different automatic service composition approaches have been proposed. Variations of forward and backward chaining methods as in [Thakkar et al. 2002], [Sheshagiri, Desjardins, Finin 2003], etc. were presented. Hierarchical Task Networks methods were proposed in such works as [Sirin et al. 2004] and [Sohrabi, Baier, McIlraith 2004]. Ontological descriptions of services can be used by reasoners for composition creation [Ankolekar et al. 2002]. Petri nets were used for service modeling and composition in [Hamadi, Benatallah 2003].

Composition methods can, but do not have to, assume that service instances are available and reachable somewhere in a network. So, if a method is not concerned with the availability of service instances it will produce abstract composition plans. On the other hand, grounding is a process of enriching composition plans with vital information that allows necessary service instances to be used during the execution process. Therefore abstract composition plans have to be grounded prior to being ready for execution. Several service composition grounding methods have been proposed, some of them are based on brokers [Chakraborty, Yesha, Joshi 2004] while some others are matching-based [Sun et al. 2009], heuristic [Liu et al. 2009], agent-based [Tang, Xu 2006] or even ontology-based [Yan, Zhijian, Guiming 2010] and [Bleul, Weise 2005]. Each one of these approaches uses different perspectives on the grounding process, thus allowing their users to fit their needs in very varied and not always interoperable ways.

1.2. Problem statement

There are numerous methods for creating and grounding service compositions. However, every particular approach cannot be an ideal solution from all points of view. Imagine a situation when a user of an SOA software system wants to use some predefined service composition parts and combine them with an output of an automatic composition method. The concept of hybrid service composition was specifically proposed in order to solve this kind of problems [Belava 2009]. This concept allows to use multiple service composition methods during the creation of a service composition plan.

A similar problem is observed in the grounding of abstract composition plans because grounding methods may vary a lot in terms of their work principles as well as optimization targets (QoS, cost, etc.). The concept of hybrid grounding tries to address this problem by utilizing different grounding methods during the grounding of one particular abstract service composition plan.

So far, several service composition platforms have been presented in scientific literature. The most important ones include SWORD [Ponnekanti, Fox 2002], METEOR-S [Aggarwal 2004], MAESTRO [Chifu et al. 2009], SPICE [Wang, Wang, Xu 2006]. However, none of them tries to solve the problem of more flexible composition or picking and using a grounding method. SWORD uses first order logic, METEOR-S adopts the Constraint Satisfaction Problem engine for producing a composition, MAESTRO is based on a particular graph method with backward chaining and SPICE uses backward chaining with branching for optimization purposes.

A variety of service composition and grounding methods, platforms and approaches has been proposed. However, none of them is perfect from every point of view. In order to solve this issue, the concept of a hybrid composition and grounding platform was developed. Hybrid service composition is a method that allows its users to combine different service composition techniques. It offers more flexibility and control of the composition process itself. Hybrid grounding is also a method that allows similar flexibility and control of the grounding process. It allows to mix and match different grounding techniques for different parts of an abstract composition plan.

2. Proposed Platform Concept

The architecture of the hybrid composition and grounding platform consists of five key modules that are cooperating together. The concept also incorporates external elements – web services. These services are used by various modules to produce and execute composition plans. Figure 1 shows the architecture of platform and data flows between different modules.

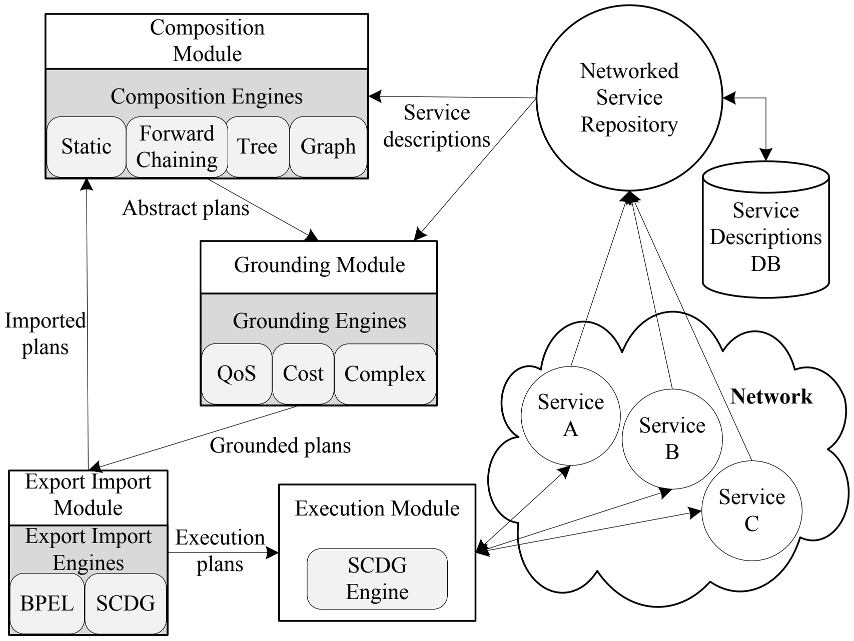


Figure 1. Architecture of hybrid composition and grounding platform

Source: own elaboration.

The static service composition engine provides the necessary functionality to combine different pieces of the composition plan that can be imported or generated by other composition engines. The static engine uses two main operations to work on composition plans: DELETE and INSERT. The "Delete" operation cuts out a specified part of a plan and "Insert" pastes one plan into another. The INSERT (2, 3, plan 1, plan 2, 5) operation scheme is presented in Figure 2. Plan1 and plan2 represent two input plans for the operation. Plan3 is the result of inserting plan2 from the first non-root node to "Service 5" node into plan1 between "Service 2" and "Service 3" nodes.

The DELETE (2, 4, plan 1) operation scheme is presented in Figure 3. Plan 2 is the result of cutting a chain of services from plan1 starting at "Service 2" and finishing at "Service 4".

To proceed further, we need to provide a definition for a service input and output type. Input or output service types in the proposed approach consist of two parts: the first – a formal description of the data format that the service accepts as input or returns as output, the second – semantic information that describes the meaning of that data.

A forward chaining service composition engine creates service composition plans by using a simple chaining algorithm similar to the one proposed in [Sheshagiri,

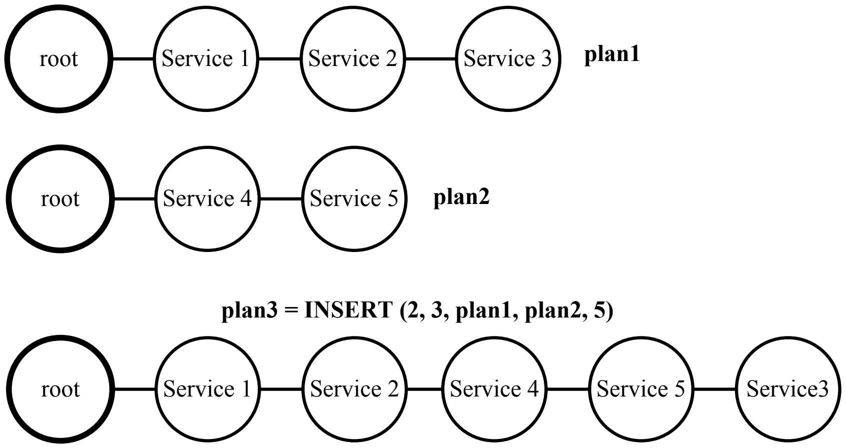


Figure 2. INSERT operation scheme

Source: own elaboration.

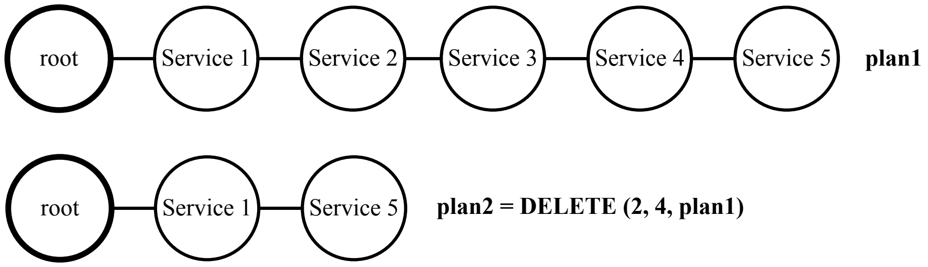


Figure 3. DELETE operation scheme

Source: own elaboration.

Desjardins, Finin 2003]. Its simplified scheme of action is to successively add new elements to the end of the plan if their input types are consistent with the previous element's output type. The general idea is to create such a chain of elements that its last element will have the desired output type.

A tree-based service composition engine creates service compositions by using a method that creates not just a chain of elements, but a tree. This method is relatively similar to forward chaining but it allows to search the produced trees and, because of that, the results of its work are more optimal than the results of simple chaining techniques.

A graph-based service composition engine uses a composition method that is similar to the one proposed in [Wang, Wang, Xu 2006]. Basically, at the beginning the composition algorithm produces a complete services dependency graph. This directed graph is created by treating abstract services as nodes in a graph and then connecting the nodes with directed arcs if one service's output type is identical to the other service's input type. Then such a graph could be processed by Dijkstra or some other pathfinding algorithms. Figure 4 presents a sample complete services dependency graph. Each node in that graph is described by its input type ("IN") and output type ("OUT").

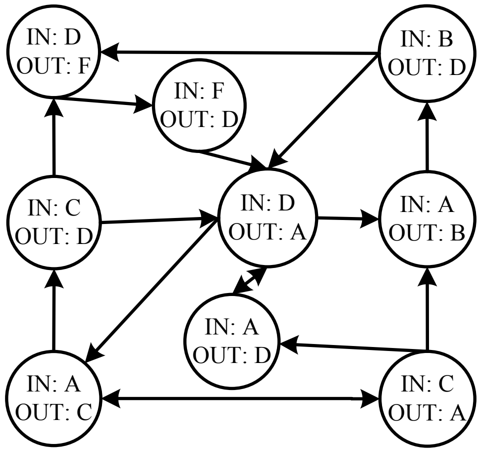


Figure 4. Example of a complete service dependency graph

Source: own elaboration.

The Grounding Module allows abstract composition plans that were produced by the composition module to be grounded. It cooperates closely with the services repository from which it gets full information profiles about service instances that are available on the network. Such a profile consists not only of the service address and input/output types but also includes additional parameters such as QoS and cost. The three grounding engines in the Grounding Module include QoS, cost and complex. QoS and cost grounding methods were chosen as sample approaches that can be successfully combined in a complex engine. There is a possibility to use and combine other grounding methods as well.

The goal of the QoS optimization engine is optimizing QoS parameters of composition plans or their parts. For example, one can request that QoS parameters for some part of the abstract composition plan have to reside between some desired maximum and minimum values. In such cases the QoS engine will look for service instances that fit the provided values best.

The cost optimization engine works similarly to the QoS engine, but it has a task to optimize the cost of composition plans or their respective parts.

The complex optimization engine allows to create a hierarchical structure of grounding preferences which lets the user apply additional optimizations in cases where the engine on a higher level of hierarchy will find several equally fitted service instances. For example, we can imagine a situation in which the cost parameter is the most important target of the composition optimization, but we would like to choose a service with the best QoS in case there are several service candidates with the same cost value.

The Export Import Module provides functionality that allows the abstract and grounded composition plan to be imported or exported from or to files. There are two export-import engines that were implemented for the proposed platform – BPEL and SCDG.

The BPEL engine is able to import [Belava 2011b] and export [Belava 2011a] composition plans that are written in a BPEL language. Not all the BPEL functionality is currently implemented, but core elements like conditionals, loops and the parallel execution of services are fully supported.

The SCDG engine allows to work with composition plans that are presented as Service Composition Directed Graphs. The SCDG is a graph-based model of service composition representation that was proposed in [Belava 2011b].

An Execution Module executes grounded service composition plans. To-date only the SCDG execution engine has been implemented, although there is a possibility to include other engines. To do that, one might also need to develop first an appropriate import-export engine.

A Networked Service Repository Module is a web service that on the one hand allows web services to be registered in it and on the other hand provides information about these services for composition and grounding modules. This module also employs a standalone database for service descriptions to be stored in it. A database engine could be either external or internal in relation to the Networked Service Repository. External database engines, however, are much faster and more reliable with large data sets and thus more preferable.

3. UseR Case Scenario

We can imagine an on-line trading system which allows its users to search for products, place orders and ultimately buy goods by entering financial and personal data into the system. There are all sorts of government regulations and industry standards for personal and financial data because of its sensitive nature. Therefore, we can be sure that some parts of the composition plans in this kind of software platforms will be predefined specifically to obey all sorts of regulations and standards. On the other hand, such systems may benefit after all from automatic or semi-automatic service composition techniques.

Hybrid service composition was proposed to solve exactly these kinds of problems by providing the necessary interoperability between different service composition methods.

3.1. System's Internal Operation – From Composition to Execution

Figure 5 presents a diagram with an example of how a service composition plan is made, grounded and executed in a system which implements the platform concept proposed in this article.

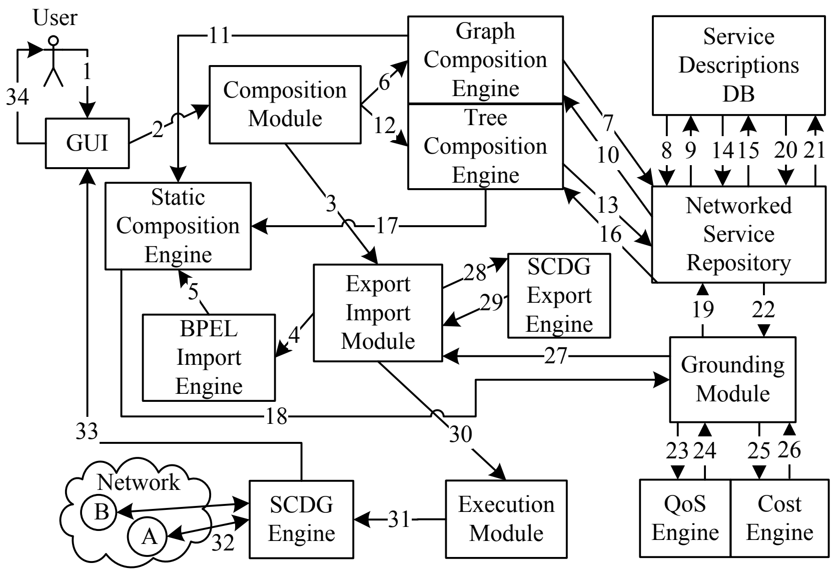


Figure 5. Service composition, grounding and execution diagram

Source: own elaboration.

1. The user provides necessary personal and financial data and the parameters of the desired products.
2. This data is delivered to a Composition Module.
3. The Composition Module sends a request to an Export Import Module to make an import of a standard-required part of the composition plan which will handle personal and financial data.
4. The Export Import Module transfers the request to a BPEL Import Engine which will actually perform the task of importing.
5. The BPEL Import Engine sends a part of the imported composition plan to a Static Composition Engine so that it can later be merged with automatically composed parts.
6. The Composition Module initiates a Graph Composition Engine and transfers composition parameters to it.
7. The Graph Composition Engine makes a request to a Networked Service Repository and asks for a list of available services types.

8. The Networked Service Repository makes an appropriate query in a Service Descriptions Database.

9. The Service Descriptions Database processes the query and sends back the results.

10. The Networked Service Repository provides the Graph Composition Engine with a list of all available services types (not instances).

11. The Graph Composition Engine sends the prepared part of the future service composition plan to the Static Composition Engine.

Steps 12…17 are similar to steps 6…11.

18. The Static Composition Engine merges all parts of the composition plan into one abstract service composition plan and delivers it to a Grounding Module for grounding.

19. The Grounding Module makes a request to the Networked Service Repository and asks it to provide a list of real-world service instances whose inputs and outputs correspond to the inputs and outputs of the services in the abstract composition plan.

Steps 20 and 21 are similar to steps 8 and 9.

22. The Networked Service Repository provides the Grounding Module with a list of required real-world service instances.

23. The Grounding Module initiates a QoS Engine and delivers the appropriate part of the plan plus the lists of service instances to it.

24. The QoS Engine grounds a part of the greater plan and sends it back to the Grounding Module.

Steps 25 and 26 are similar to steps 23 and 24.

27. The grounded composition plan is delivered to the Export Import Module.

28 The Export Import Module initiates a SCDG Export Engine and provides it with a grounded composition plan.

29. An exported composition plan is delivered back to the Export Import Module.

30. The Export Import Module sends the exported composition plan to the Execution Module for plan execution to be made.

31 The Execution Module initiates a SCDG Execution Engine and provides it with a composition plan.

32. The SCDG Execution Engine executes the composition plan.

33. Plan execution results are delivered to the interface.

34. The interface renders the acquired results and presents them to the user.

3.2. A Closer Look at Composition and Grounding

Figure 6 presents a visualization of an abstract composition plan which deals with sensitive personal and financial data provided by the user. There are several service calls in it: “AssignUniqueID” – assigns a unique ID number to a user-provided data set, “Encrypt” – encrypts the data set, “Archive” – archives the previously encrypted data, “ValidateData” – makes appropriate validations of the user-provided data.

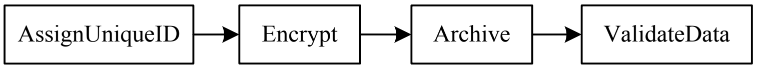


Figure 6. Abstract composition plan in a BPEL language with predefined service calls

Source: own elaboration.

The automatic generation of the second and third parts of a composition plan was done by graph and tree composition engines. The graph-based engine composed the part of the plan responsible for product finding, selecting and placing an order in a system. The tree-based engine composed the part responsible for the processing of the order that had been placed earlier.

Figure 7 presents a visualization of a complete services dependency graph of all registered types of web services that were registered in the Networked Service Repository. That exact graph was generated by a Graph Composition Engine during the composition process itself and visualized by the visualization functionality of the software platform. All service type IDs were automatically generated by the Networked Services Repository. Each of these IDs consisted of service input type name, “|” character and service’s output type name.

The left part of Figure 8 presents a visualization of an abstract plan that was generated by a Graph Composition Engine. “ValidationResultID” is an output type of the last service call in a predefined part of the service composition which was imported from a BPEL file, so it was passed to the Graph Composition Engine as a desired input type. “OrderID” type is a type which corresponds to the output type of the order creation service, so it was passed to the composition engine as a desired output type of the composition.

The subsequent steps of a plan generated by the Graph Composition Engine are as follows:

1. “ValidationResultID|OfferPack” represents an automatic wide search of possible products on the client’s request.

2. “OfferPack|FilteredOfferPack” represents automatic filtering of the previously found products.

3. “FilteredOfferPack|ClientApproval” represents the client’s acceptance of a product offer.

4. “ClientApproval|OrderID” represents generating an order for the offer that had already been accepted.

The right part of Figure 8. presents a visualization of an abstract plan generated by a Tree Composition Engine. “OrderID” type was passed to the composition engine as a desired input type because it has to be the same as the output type of a Graph Composition Engine’s work result. “ClientNotificationID” represents the result of client notification which always happens after an order is processed, so it was passed to the Tree Composition Engine as a desired output type.

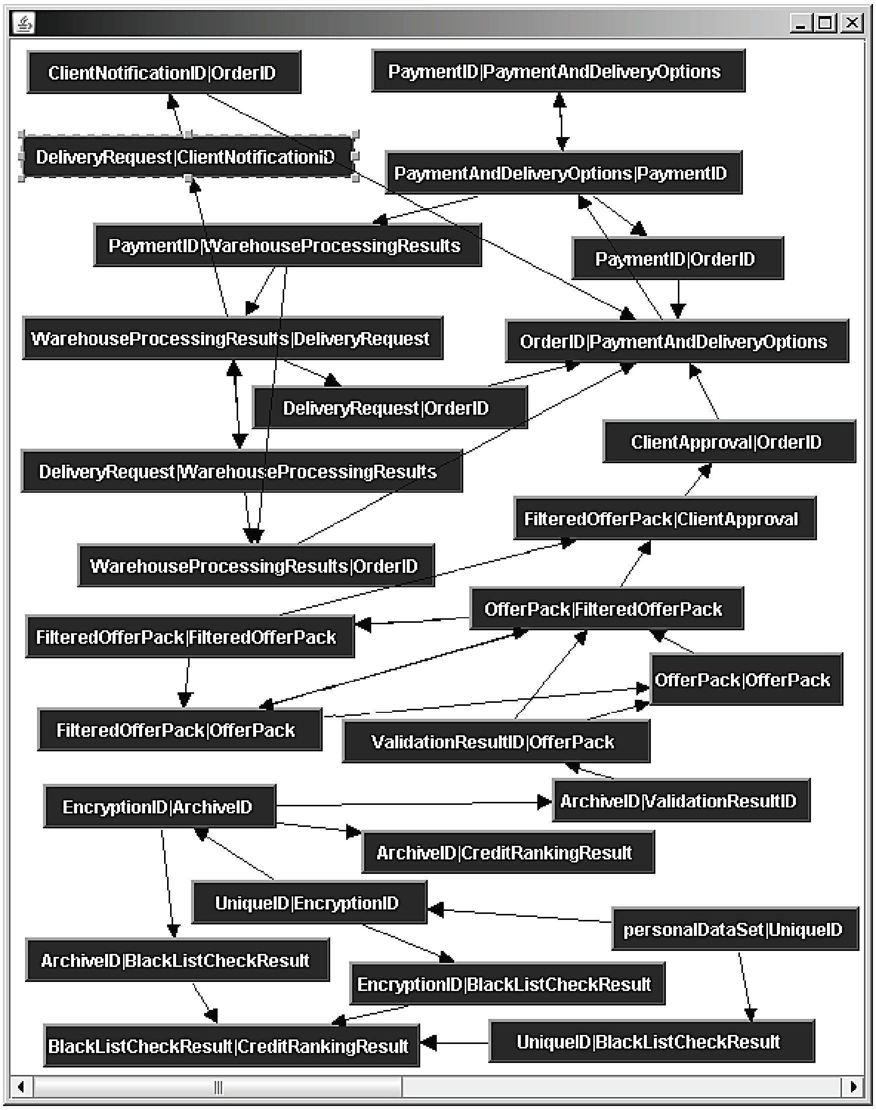

ClientNotificationID|OrderID
PaymentID|PaymentAndDeliveryOptions
DeliveryRequest|ClientNotificationID
PaymentAndDeliveryOptions|PaymentID
PaymentID|WarehouseProcessingResults
PaymentID|OrderID
WarehouseProcessingResults|DeliveryRequest
OrderID|PaymentAndDeliveryOptions
DeliveryRequest|OrderID
ClientApproval|OrderID
DeliveryRequest|WarehouseProcessingResults
FilteredOfferPack|ClientApproval
WarehouseProcessingResults|OrderID
OfferPack|FilteredOfferPack
FilteredOfferPack|FilteredOfferPack
OfferPack|OfferPack
FilteredOfferPack|OfferPack
ValidationResultID|OfferPack
EncryptionID|ArchiveID
ArchiveID|ValidationResultID
ArchiveID|CreditRankingResult
UniqueID|EncryptionID
personalDataSet|UniqueID
ArchiveID|BlackListCheckResult
EncryptionID|BlackListCheckResult
BlackListCheckResult|CreditRankingResult
UniqueID|BlackListCheckResult

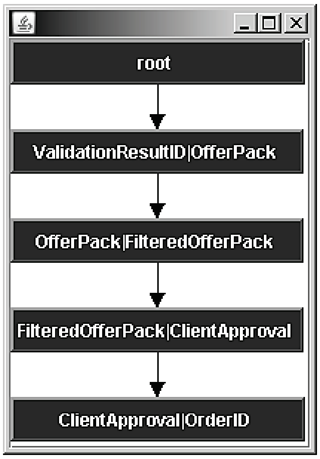
Figure 7. Complete service dependency graph for Service Repository

Source: own elaboration.

The subsequent steps of a plan generated by the Tree Composition Engine are as follows:

1. "OrderID|PaymentAndDeliveryOptions" represents a user's process of choosing payment and delivery options for a created order.

2. "PaymentAndDeliveryOptions|PaymentID" represents the act of payment for delivery by a client.


root
ValidationResultID|OfferPack
OfferPack|FilteredOfferPack
FilteredOfferPack|ClientApproval
ClientApproval|OrderID

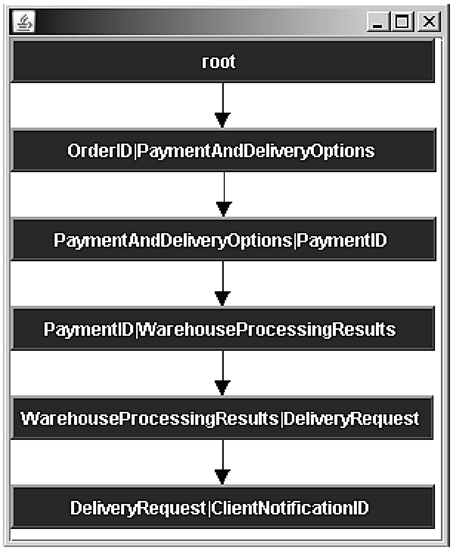

root
OrderID|PaymentAndDeliveryOptions
PaymentAndDeliveryOptions|PaymentID
PaymentID|WarehouseProcessingResults
WarehouseProcessingResults|DeliveryRequest
DeliveryRequest|ClientNotificationID

Figure 8. Graph (left) and Tree (right) Composition Engines work results

Source: own elaboration.

3. “PaymentID|WarehouseProcessingResults” represents all background warehousing processing, such as searching for the warehouse nearest to the client, scheduling product pickup from the shelf, packing etc.

4. “WarehouseProcessingResults|DeliveryRequest” represents creating a delivery request to a logistics company which will actually deliver the products to the customer.

5. “DeliveryRequest|ClientNotificationID” represents the client notification process during which the client receives information about the delivery and other order related matters.

All three parts of a complete abstract service composition plan were merged after they were created or imported by the corresponding engines. After that the complete plan was divided into three grounding areas and grounded in a hybrid mode.

The first grounding area consisted only of the steps from the first part of the plan which had been imported from the BPEL. Because this part is very important and regulated by government and industry standards, it was grounded only by a QoS Engine which was tuned to select the best available service instance, no matter the cost.

The second grounding area was defined as a steps chain from “ValidationResultID|OfferPack” service up to “OrderID|PaymentAndDeliveryOptions”. The main grounding engine for that area was the Cost Engine and the second one was the QoS engine. The Cost Engine, however, was configured to choose not the absolutely best service from a variety of the available ones, but a range of acceptable services within a provided distance from the best one. The additional grounding engine for the second area was the QoS Engine which was able to choose the service instance with the best cost from the range of the previously selected ones by the Cost Engine.

The third grounding area was defined as a steps chain from "PaymentAndDeliveryOptions|PaymentID" up to "DeliveryRequest|ClientNotificationID". It was grounded similarly to the second part, but the difference was that the main grounding engine was the QoS Engine and the second one was the Cost Engine.

Figure 9 presents a visualization of a grounded composition plan. The only difference between the visualizations of the abstract and grounded composition plans

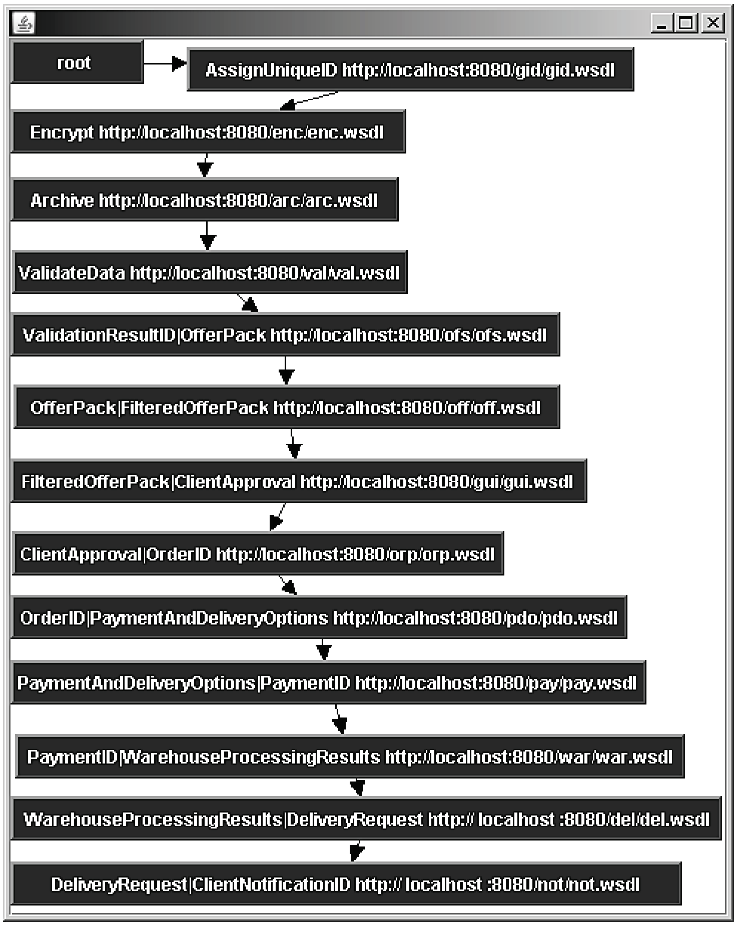

root
AssignUniqueID http://localhost:8080/gid/gid.wsdl
Encrypt http://localhost:8080/enc/enc.wsdl
Archive http://localhost:8080/arc/arc.wsdl
ValidateData http://localhost:8080/val/val.wsdl
ValidationResultID|OfferPack http://localhost:8080/ofs/ofs.wsdl
OfferPack|FilteredOfferPack http://localhost:8080/off/off.wsdl
FilteredOfferPack|ClientApproval http://localhost:8080/gui/gui.wsdl
ClientApproval|OrderID http://localhost:8080/orp/orp.wsdl
OrderID|PaymentAndDeliveryOptions http://localhost:8080/pdo/pdo.wsdl
PaymentAndDeliveryOptions|PaymentID http://localhost:8080/pay/pay.wsdl
PaymentID|WarehouseProcessingResults http://localhost:8080/war/war.wsdl
WarehouseProcessingResults|DeliveryRequest http:// localhost :8080/del/del.wsdl
DeliveryRequest|ClientNotificationID http:// localhost :8080/not/not.wsdl

Figure 9. Grounded composition plan

Source: own elaboration.

are the URL addresses of the WSDL files in every step of the composition. These addresses unequivocally correspond to real-world service instances due to that fact that the data in each WSDL file describes a concrete service instance.

The execution of a service composition plan was made with the use of an Execution Module which was making service calls to appropriate instances by their URLs.

4. Implementation Details

4.1. Package structure overview

The described platform was implemented in Java 6 programming language. Figure 10 presents its package diagram. The Service Composition Directed Graph package consists of the code which implements the SCDG data structure and an API for operating on it. Such an approach enables to reuse this vital code across all the

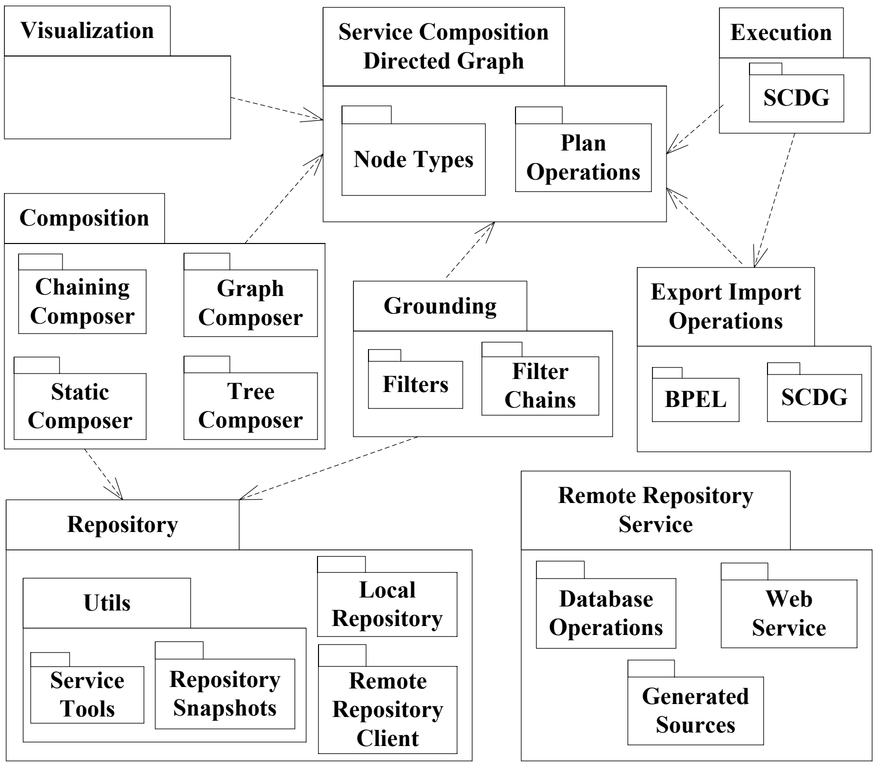


Figure 10. Platforms package diagram

Source: own elaboration.

platform in Composition and Grounding and Export Import modules in an easy and consistent manner. This package consists of two sub-packages: Node Types which implements the nodes hierarchy in the SCDG data structure, and Plan Operations which implements all the APIs for such plan manipulation tasks as adding a node or getting all arcs for a specific node.

A composition package implements all composition logic exactly and consists of four sub-packages. Each of these sub-packages implements a different service plan composition method. The grounding package provides all the grounding logic and consists of two sub-packages which implement grounding engines. The QoS and Cost grounding was implemented with the usage of grounding filters in the Filters sub-package. Basically, these filters can choose appropriate service instances from a set of available service instances provided by the Repository Module based on parameters given by the user. Complex grounding was implemented with the use of a Filter Chains sub-package which allows to create specific filter chains, thus providing required functionality to implement the Complex Grounding Engine. The repository package implements valuable tools which allow other modules to use the Networked Repository Service module. The Utils sub-package implements various classes for working with repository snapshots. Local and Remote Repository sub-packages are used as an interface to the Networked Repository Service. The Local sub-package acts as a local repository which can be used if the Networked Repository Service is unavailable or for conducting simple tests. The Remote Repository Client package implements all communications with the Networked Repository Service. The Export Import Operations package implements the functionality of importing and exporting BPEL and SCDG composition plans to and from the XML files. The Execution package implements necessary logic and communication methods for the execution of SCDGs. The Visualization package generates all the visualizations of grounded and abstract composition plans. The Remote Repository Service package is a package that implements a standalone Networked Repository Service module. Its sub-packages are Database Operations, Web Service and Generated sources. The Database Operations sub-package implements all database interactions logic, like adding new service descriptions to the database, selecting or deleting them. The Web Service sub-package implements a specific functionality that allows the Networked Repository Module to operate as a Web Service in a network. Finally, the Generated Sources package consists of auto-generated sources which are helpful when processing SOAP requests and responses that the Networked Repository Service has to manipulate during interactions with other modules or services.

4.2. A Closer look at main packages

Figure 11 presents a class diagram for a Service Composition Directed Graph package.

This package implements the SCDG data structure and an API for manipulating it. The package heavily relies on JgraphT 0.8 library which provides some abstractions

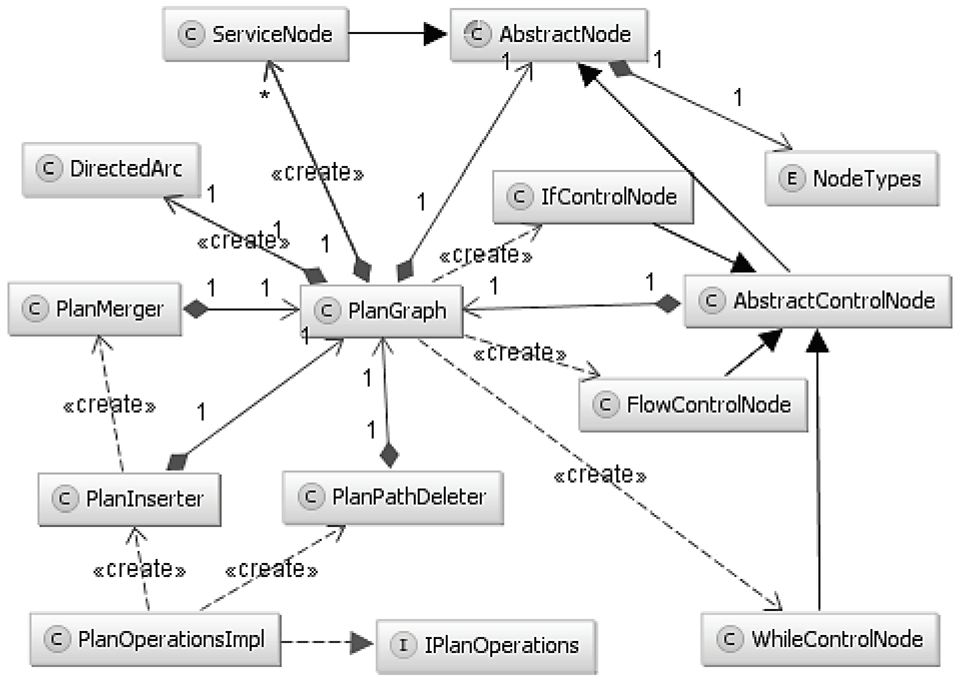


Figure 11. Service Composition Directed Graph package class diagram

Source: own elaboration.

and algorithms for the manipulation of graph structures. All the node types were implemented as descendants of an abstract AbstractNode class. All service nodes in the SCDG structure were implemented by extending the AbstractNode class, however control nodes such as Flow, While and If extend an abstract class called AbstractControlNode which is also a descendant of the ControlNode class. NodeTypes enumeration was introduced for the ease of working with references to the AbstractNode class. The arcs in the SCDG data structure were implemented in the DirectedArc class which extends the DefaultEdge class from the JgraphT library. Different operations on the SCDG structures, such as the deletion of a path in a plan or merging different plans, were implemented in the PlanOperationsImpl class which uses PlanInserter and PlanPathDeleter classes.

Figure 12 presents a class diagram for the Composition package which implements service composition algorithms.

The Forward Chaining, Tree and Graph composition engines implement the IDynamicComposer common interface which unifies the API for all of them. The Static engine, however, implements only the IStaticComposer interface because a static engine provides a different functionality than dynamic ones do. Dynamic composers operate on given sets of composition requirements plus data about available

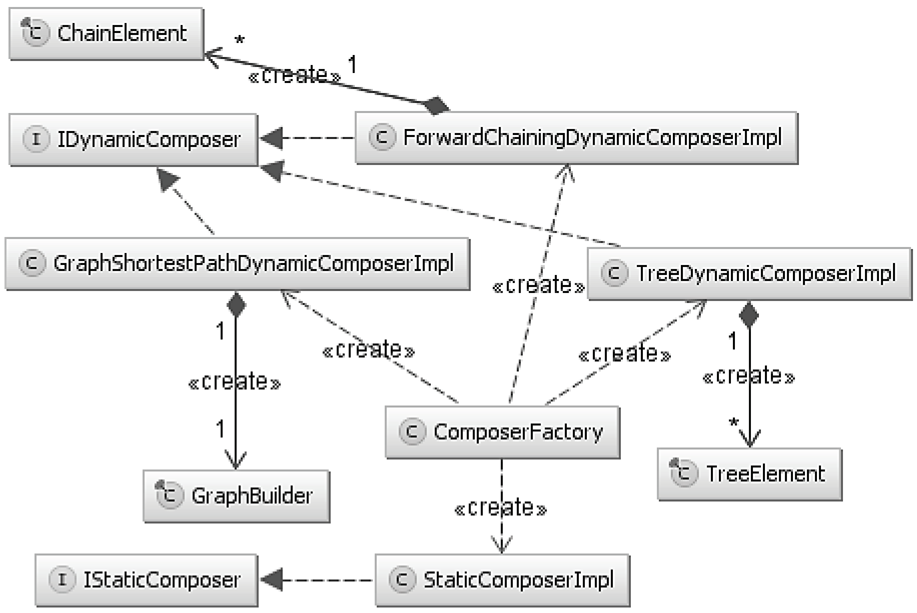


Figure 12. Composition package class diagram

Source: own elaboration.

abstract services, while a static composer operates only on composition plans and arranges them in a required manner. Dynamic composers also have helper classes for their internal needs, like the ChainElement class which represents a chain element for the Forward Chaining engine and provides some valuable methods to operate on itself. There is also the TreeElement class which is a helper class for the Tree engine and the GraphBuilder class which is helpful to the Graph engine and provides functionality for building and operating on complete services dependency graphs.

Figure 13 presents a class diagram for the Grounding package which implements all grounding logic in the described platform.

As was mentioned earlier in this article, grounding engines were implemented with a concept of filters and filter chains. This was done bearing in mind a greater code clarity, reusability and the ability to add more grounding engines by implementing appropriate filters. IServiceSetGroundingFilter interface is the main interface for all grounding filter implementation classes. This interface defines methods that should be implemented by filter classes which will be used in the grounding process. The QosServiceSetGroundingFilterImpl and CostServiceSetGroundingFilterImpl classes implement the above-mentioned interface and work their parts in the QoS and Cost grounding engines. The GroundingFilterChain and ServiceSetGroundingFilterChain classes implement a vast functionality of a filter chain concept, too. During the groun-

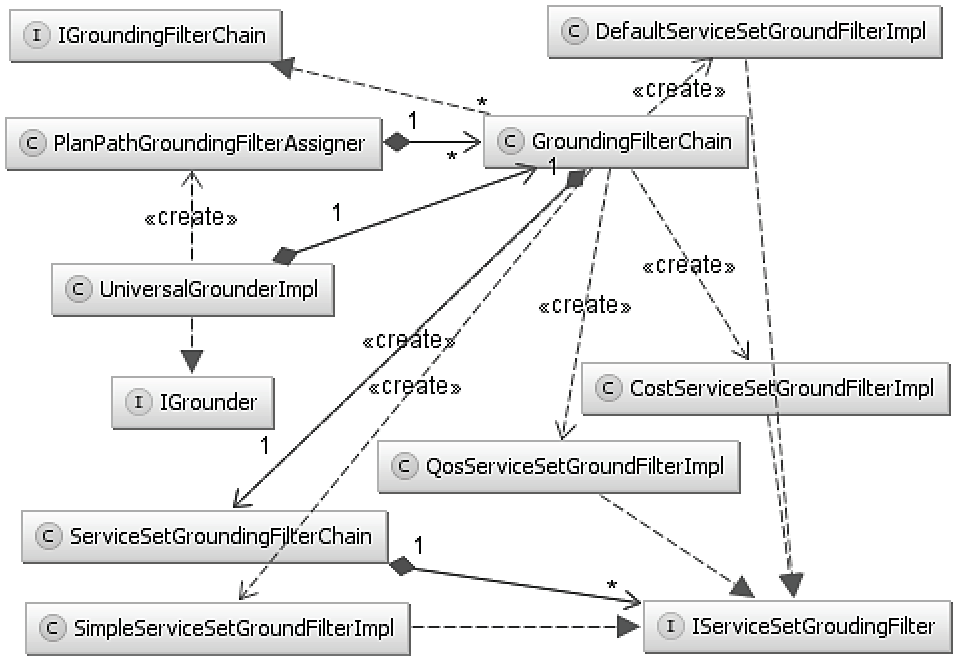

IGroundingFilterChain
DefaultServiceSetGroundFilterImpl
PlanPathGroundingFilterAssigner
GroundingFilterChain
UniversalGrounderImpl
IGrounder
CostServiceSetGroundFilterImpl
QosServiceSetGroundFilterImpl
ServiceSetGroundingFilterChain
SimpleServiceSetGroundFilterImpl
IServiceSetGroudingFilter
«create»
1
*

Figure 13. Grounding package class diagram

Source: own elaboration.

ding procedure, filters can be stored and arranged into chains by the GroundingFilterChain object which uses the ServiceSetGroundingFilterChain class objects for holding and applying these filters. Thus the arranged filter chains can be assigned to specific parts of a plan by a PlanPathGroundingFilterAssigner class object. Such an approach allows the described platform to ground different parts of composition plans by different grounding engines. The UniversalGrounderImpl class implements the IGrounder interface and provides an API for the whole grounding module. There are also two utility filter classes in this package: SimpleServiceSetGroundFilterImpl and DefaultServiceSetGroundFilterImpl. The first represents a simple grounding filter which only filters services with appropriate input and output types and does no other work. The second one implements a specific filter that is used only for the purpose of clearing filter chains.

4.3. Tools, libraries and protocols used

Several different tools, libraries and protocols were used during the implementation of the described platform. First of all, the platform was written to effectively work with Web Services. The Spring Web Services 2 framework was used to create sample test services and the Networked Service Repository since it is a web service, too. All of them were implemented as servlets that can run in a proper servlet container. Apache Tomcat 7 was responsible for that task and it hosted all the services along with the Networked Service Repository. A SOAP protocol over HTTP was used to handle all the communication between web services (including the Networked Service Repository) and the modules that were communicating with them (the Execution, Grounding and Composition modules). The hibernate version 4 object-relational mapping framework was used to organize the storage of service related information in a convenient manner. The H2 version 1.3 database was used as a store facility for all service related data. H2 is an in-memory database, so it is not optimized for real-life projects. However, for projects of a prototype nature, such as presented in this paper, it is appropriate and also easy to work with. The SCDG data structure was implemented with the extensive usage of the JgraphT 0.8 library. This library also provided some valuable graph algorithms for the Graph Composition Engine. The handling of XML data across all platform modules was performed with the use of the Jdom 1.3 library. The JGraph 5 library was used to draw the visualizations of service composition plans in the SCDG format such as in Figures 7, 8 and 9.

5. Conclusion and Future Work

This article presents a novel approach to a software system that allows its users to combine different service composition and grounding methods. Such a possibility enables users to control the composition and grounding processes in a different and more powerful way, thus allowing them to create better suited abstract and grounded composition plans.

The proposed approach was implemented, tested and verified. The verification process revealed that hybrid composition and hybrid grounding techniques are viable tools and can be implemented and used in real world applications.

The main implication of the presented work is the fact that users of software platforms that implement the proposed approach will have more flexibility and control over service composition and grounding processes. Many composition and grounding methods have been proposed, yet each of them is different and may not suit all the needs of the endpoint customer. Furthermore, to satisfy all the upcoming and even unknown business needs, software systems must allow changes to be introduced in them. The ability to choose and combine different service composition and grounding methods addresses these problems by enabling users to select and merge optimal methods for their needs.

There are also two main directions of the upcoming work for the proposed concept. The first one is a hybrid execution of grounded plans. The combined usage of different execution engines might bring some additional features, since these engines might employ different approaches and thus be valuable from different points of view. The second direction of the studies has to be made in the field of dynamic composition, grounding and execution methods. Such methods can be very desirable e.g. in software platforms where the fault-tolerance level of services is low or the environment itself may constantly be changing.

References

Aggarwal R., *Constraint driven Web Service composition in METEOR-S*, Proceedings IEEE International Conference on Services Computing, Sep. 2004, pp. 22-30.

Ankolekar A., Burstein M., Hobbs J., Lassila O., Martin D., *DAML-S: Web Service description for the Semantic Web*, Proceedings International Semantic Web Conference (ISWC) 2002, June 2002, pp. 348-363.

Belava L., *Algorytm konwersji skierowanego grafu kompozycji serwisów do planów kompozycji serwisów webowych w języku BPEL*, „Automatyka" 2011a, vol. 15/2, pp. 71-80.

Belava L., *Koncepcja hybrydowej kompozycji usług w środowisku SOA*, „Automatyka" 2009, vol. 13/2, pp. 189-197.

Belava L., *Transforming BPEL service composition into a service composition directed graph for better composition plan management*, Proceedings 25th European Conference on Modelling and Simulation, June 2011b, pp. 424-429.

Bleul S., Weise T., *An ontology for quality-aware service discovery*, Proceedings First International Workshop on Engineering Service Compositions, Dec. 2005, pp. 35-42.

Chakraborty D., Yesha Y., Joshi A., *A distributed service composition protocol for pervasive environments*, Proceedings 2004 IEEE Wireless Communications and Networking Conference, Mar. 2004, pp. 2575-2581.

Chifu V., Salomie I., Riger A., Radoi V., *A graph based backward chaining method for Web Service composition*, Proceedings IEEE 5th International Conference on Intelligent Computer Communication and Processing, Aug. 2009, pp. 237-244.

Hamadi R., Benatallah B., *Petri Net-based model for Web Service composition*, Proceedings 14th Australasian database conference on Database technologies, 2003, pp. 191-200.

Liu D., Shao Z., Yu C., Chen D., Fan G., *A heuristic QoS-aware service selection approach to Web Service composition*, Proceedings 8th IEEE/ACIS International Conference on Computer and Information Science, June 2009, pp. 1184-1189.

Ponnekanti S., Fox A., *SWORD: A developer toolkit for Web Service composition*, Proceedings 11th International WWW Conference, May 2002.

Sheshagiri M., Desjardins M., Finin T., *A planner for composing services described in DAML-S*, Proceedings AAMAS Workshop on Web Services and Agent-based Engineering, July 2003.

Silva E., Pires L.F., Sinderen M., *An algorithm for automatic service composition*, Proceedings 1st International Workshop on Architectures, Concepts and Technologies for Service Oriented Computing, July 2007, pp. 65-74.

Sirin E., Hendler J., Parsia B., *Filtering and selecting semantic Web Services with interactive composition techniques*, "IEEE Intelligent Systems" 2004, vol. 19, pp. 42-49.

Sirin E., Hendler J., Parsia B., *Semi-automatic composition of Web Services using semantic descriptions*, Proceedings Web Services: Modeling, Architecture and Infrastructure workshop in ICEIS2003, Apr. 2003, pp. 17-24.

Sirin E., Parsia B., Wu D., Hendler J., Nau D., *HTN planning for Web Service composition using SHOP2*, "Web Semantics: Science, Services and Agents Journal" 2004, vol. 4, pp. 377-396.

Sohrabi S., Baier J., McIlraith S., *HTN planning with preferences*, "Web Semantics: Science, Services and Agents Journal" 2004, vol. 4, pp. 377-384.

Sun S., Tang X., Yan X., Chen D., *A symmetric matchmaking engine for Web Service composition*, Proceedings 15th International Conference on Parallel and Distributed Systems, Dec. 2009, pp. 810-814.

Tang J., Xu X., *An adaptive model of service composition based on policy driven and multi-agent negotiation*, Proceedings 5th International Conference on Machine Learning and Cybernetics, Aug. 2006, pp. 113-118.

Thakkar S., Knoblock C., Ambite J., Shahabi C., *Dynamically composing Web Services from on-line sources*, Proceedings AAAI-2002 Workshop on Intelligent Service Integration, July 2002.

Wang Y., Wang H., Xu X., *Web Services selection and composition based on the routing algorithm*, Proceedings 10th IEEE International Enterprise Distributed Object Computing Conference Workshops, Oct. 2006, pp. 69-73.

Yan H., Zhijian W., Guiming L., *A novel Semantic Web Service composition algorithm based on QoS ontology*, Proceedings 2010 International Conference on Computer and Communication Technologies in Agriculture Engineering, June 2010, pp. 166-168.

KU PLATFORMIE HYBRYDOWEJ KOMPOZYCJI I GRUNTOWANIA USŁUG SIECIOWYCH

Streszczenie: Artykuł przedstawia metodę hybrydowej kompozycji usług sieciowych, metodę hybrydowego uziemiania abstrakcyjnych planów kompozycji oraz podejście do tworzenia platformy programowej je realizującej. W sposób szczególny zaprezentowano w nim również architekturę powstałej platformy, a także opisano jej moduły składowe.

Słowa kluczowe: SOA, kompozycja usług sieciowych, gruntowanie usług sieciowych.