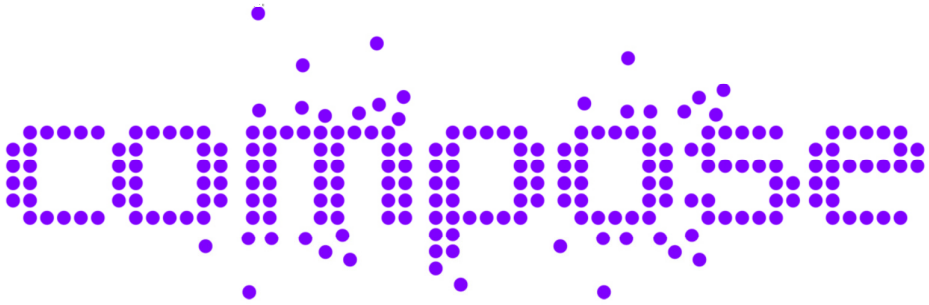

Collaborative Open Market to Place Objects at your Service

D3.1.3.1

Assisted Service Composition Engine – First prototype

Project Acronym	COMPOSE	
Project Title	Collaborative Open Market to Place Objects at your Service	
Project Number	317862	
Work Package	WP3.1 Service Management	
Lead Beneficiary	OU	
Editor	Carlos Pedrinaci	OU
Reviewer	Daniel Schreckling	UNI PASSAU
Reviewer	Iacopo Carreras	UH
Reviewer	Benny Mandler	IBM
Dissemination Level	PU	
Contractual Delivery Date	30/04/2014	
Actual Delivery Date	30/04/2014	
Version	V1.0	

Abstract

The development of applications for the Internet of Things and Services is expected to be characterised by the need to reuse and integrate various sensors, actuators, and remote services. Those components of future Internet of Things applications will have to be adequately discovered among an overwhelming set of potential sources of data and functionality, and they will have to be combined in an effective yet seamless way.

To this end, the COMPOSE platform provides an Assisted Service Composition Engine which is in charge of supporting application developers in building such applications. The engine aims to provide automated support for developers that can, given the semantics of the data available and the semantics of the data required to be obtained, automatically generate possible compositions. The first prototype exploits the semantic annotations of the data model of services and operations (i.e., their inputs and outputs) in order to determine possible workflows that are semantically compatible from a dataflow perspective. Work on this prototype has focussed on providing high-performance automated composition even when dealing with distributed, large-scale service registries, handling thousands of services with a response time of a second even when using standard desktop hardware.

This deliverable provides a general description of the component devised, presenting notably its main contributions over the state of the art, its architecture, and its integration with further components from the COMPOSE platform. The deliverable additionally includes performance evaluation results that indeed show that currently this composition engine provide most efficient composition support as necessary to cope with large-scale Internet of Things and Services scenarios.

Document History

Version	Date	Comments
V0.1	27/03/2014	Core deliverable content.
V0.2	10/04/2014	Included details on security mechanism integration.
V0.3	15/04/2014	Overall polishing.
V0.4	25/04/2014	Addressed reviewer comments
V0.5	28/04/2014	Addressed reviewer comments
V1.0	30/04/2014	Addressed reviewer comments

Table of Contents

List of Figures

List of Tables

Acronyms

Acronym	Meaning
COMPOSE	Collaborative Open Market to Place Objects at your Service
SOA	Service Oriented Architecture
API	Application Programming Interface
OWL	Web Ontology Language
I/O	Input/Output
REST	Representational state transfer
DAG	Directed Acyclic Graph
WSC	Web Service Challenge
PDDL	Planning Domain Definition Language
GUI	Graphical User Interface

1 Introduction

A fundamental tenet of COMPOSE, and Service-Oriented Architectures in general, is facilitating the development of complex software and applications by combining pre-existing, possibly distributed, software components called services [1]. The resulting software, referred to in COMPOSE as *Composite Services*, thus reuses existing functionality to provide added-value solutions. In a nutshell, Composite Services may benefit from the data and functionality exposed by sensors, actuators or remote services, e.g., Web Services and Web APIs, to enable the creation of advanced applications.

The process of combining services to create an application is often referred to as *Composition* [1], [2]. Given the potential complexity and effort required for performing this activity, notably when vast amounts of services are available, dedicated software is typically provided for assisting developers in composing new applications. Supporting software includes both manual and automated systems that may assist in the creation of compositions at design-time and/or at run-time[2]. Manual solutions include typically a tool with a simple Graphical User Interface allowing developers to easily chain services through a simple point & click interface. Automated solutions on the other hand apply advanced techniques, e.g., Artificial Intelligence planning or graph search algorithms, to automatically generate plausible compositions.

The Assisted Service Composition Engine described in this deliverable is an automated composition engine that also benefits from a friendly end-user interface (see WP6) so that developers can trigger the generation of compositions and ultimately refine and adapt them to their liking and requirements. In this manner, developers aiming to create applications over COMPOSE can quickly and easily generate service compositions without losing the ability to manually fine-tune their applications if necessary.

The first version of the composition engine described in this deliverable, is focused on supporting the generation of compositions with a configurable level of semantic compatibility of dataflow—from directly executable to skeletal plans that may require performing some mediation—exploiting registries with thousands of services with sub-second average response time. This work builds upon state of the art solutions and evolves them towards high-performance solutions in highly distributed settings as necessary for the Internet of Things.

In the remainder of this deliverable we first describe the overall approach followed by the composition engine highlighting notably the advances provided over the state of the art. We describe the architecture of the engine and its main components. We include as well the API exposed by the engine so that other components and applications can use it and pay particular attention to its integration with the discovery engine and the requirements it imposes on it with respect to its performance. Finally, we present performance evaluation results for the engine as well as a comparison with state of the art solutions and we conclude by highlighting some of the main features that are planned for the final prototype.

2 Overall Approach

Despite the appealing characteristics of service-orientation principles and technologies, the systematic development of service-oriented applications is considerably hampered by the need for software developers to devote significant labour to discovering sets of suitable services, understanding their functionality and interfaces, developing software that overcomes their inherent data and process mismatches, and finally combining them into a complex composite process.

Over the years, service composition has received much attention both from industry and academia and as a result a plethora of tools have been produced ranging from mere graphical support to completely automated solutions [1]-[3]. Automated composition solutions have received most attention given their potential benefits. Most of the work in this regard has been approached as a planning task [2]-[4], which benefits from the formal specification of Web services inputs, outputs, preconditions, and effects to generate suitable compositions [5]-[9]. Despite the wealth of algorithms and implementations described in the literature, it is considerably difficult to find robust and scalable solutions one could seamlessly adopt and reuse within the software development stack.

On the one hand, most of the engines have typically focused on dealing with considerably complex problem and service descriptions including expressive preconditions and effects. While advanced, these engines have often been developed as a proof of concept and have paid less attention to the scalability and robustness of the approach. On the other hand, planning based solutions, as they have been developed thus far, rely on two main assumptions that are difficult to ensure—especially as the scale of the deployment envisaged grows. First and foremost, these techniques rely on the existence of complex preconditions and effects that are seldom found in semantic Web service descriptions due to their complexity [10]. In fact, out of all the descriptions of semantic Web services found on the Web, less than 5% include preconditions and effects [10]. Second, these engines rely, for the most part, on loading the entire set of services available in memory. This last assumption presents obvious limitations from a scalability point of view and, most importantly, it requires complete access to the data held by the registry or registries used, which may well go against the interests of the registry providers.

While research in the area has typically evolved towards dealing with increasing complex service and problem descriptions, in developing this composition engine we have focused instead on providing a solution that is scalable and efficient in the scenarios one is likely to encounter in the Internet of Things. That is in scenarios where thousands of heterogeneous services seldom described by means of expressive preconditions and effects, are exposed on the Web through a number of distributed third-party registries. The second prototype will introduce the use of more expressive preconditions and effects in order to benefit from and honour security-specific axioms that will be generated by the WP5 infrastructure. This more advanced scenario will indeed benefit from this first phase of very fast generation of possible compositions, prior to the ulterior evaluation of complex conditions, which would only take place over a very small subset of services in order to retain good computational performance.

In addition to the composition engine itself, this work also provides contributions to the areas of composition and discovery from a conceptual perspective providing more in depth understanding on the intertwining between service composition and discovery.

Notably, thus far, and despite the obvious dependencies between discovery and composition, research in both areas has evolved largely independently. Composition engines, in part due to them approaching composition as a planning problem, have typically replicated discovery functionality internally and have therefore not exploited the latest advances in discovery. Besides, as previously introduced, they rely on having all service descriptions available in local memory, which is too strong an assumption. Discovery solutions, on the other hand, have traditionally approached discovery as an activity essentially triggered sporadically by a human. As a consequence response time has hardly been a concern and the programmatic interfaces exposed by discovery engines are usually thought for one-of requests and are hardly optimised for frequent, high throughput interactions, as necessary to support efficient composition.

The work presented in this deliverable has been carried out in close collaboration with the discovery engine and has yielded as well valuable contributions on the integration and dependencies between discovery and composition. This includes the definition of a generic composition framework that embraces (rather than replaces) discovery engines, the definition of a generic programmatic interface that discovery engines should implement, and provides tangible and specific performance requirements that discovery engines should exhibit.

2.1 Graph-Based Service Composition

Service Composition is the process of finding a composition of viable service invocations that, given a set of requirements and constraints can lead to the desired or required outcome. The requirements and constraints may range from the semantics of the data available and required, to a set of constraints or preferences over non-functional properties (e.g., services should be secure).

In the first prototype of the Assisted Service Composition Engine, the specification of the service composition problem is limited to the semantics of the data available and required. In particular, given the semantics of the data available for processing and the semantics of the data that we want to obtain, the Assisted Service Composition Engine automatically explores the set of potential compositions given the services known to the COMPOSE infrastructure and provides a ranked set of viable compositions. In this first prototype the focus lies on the generation of workflows with a required and configurable compatibility in terms of the semantics of the data flowing between the services/operations constituting the composition.

Figure 1 shows an example of a service composition including both the requested inputs and outputs. In particular, in this example the user is looking for ways to obtain the weather for his or her actual location, given his or her actual IP address and some login credentials. On the basis of this request, the composition engine is in charge of figuring out if there is a possible sequence of service invocations that could lead from the provided input data, to the required output data. In the example, a potential process composed of 3 services, e.g., *WhoisService*, *WeatherAuthService*, and *WeatherService*, together with the corresponding dataflow definition

is found. The process in the figure exploits the semantics of the data exchanged in order to ensure that services are invocable. Notably, the engine exploits the fact that a *Country isA Place* and therefore knows that *WeatherService* would be invocable using directly the *Country* obtained in the previous invocation of the *WhoisService*.

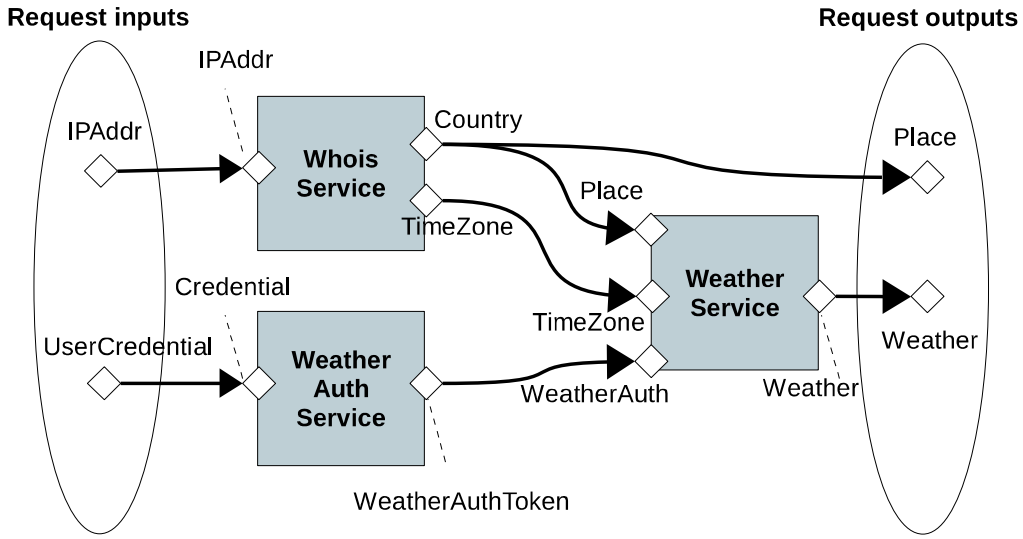

Figure 1. Example of a Service Composition.

An important part of the process of defining the data flow and figuring out the potential sequence of invocable services involves checking the compatibility between inputs and outputs of services. This process often referred to as matchmaking is typically contemplated in semantic service discovery activities and generally includes different degrees of compatibility [11]:

- **Exact**: the output of a service is of a semantic type that is equivalent to that of the input of the subsequent service.
- **Plugin**: the output of a service is a sub-concept of the input of the subsequent service.
- **Subsume**: the output a service is a super-concept of the input of the subsequent service.
- **Fail**: none of the previous matches are found between the service's output and inputs.

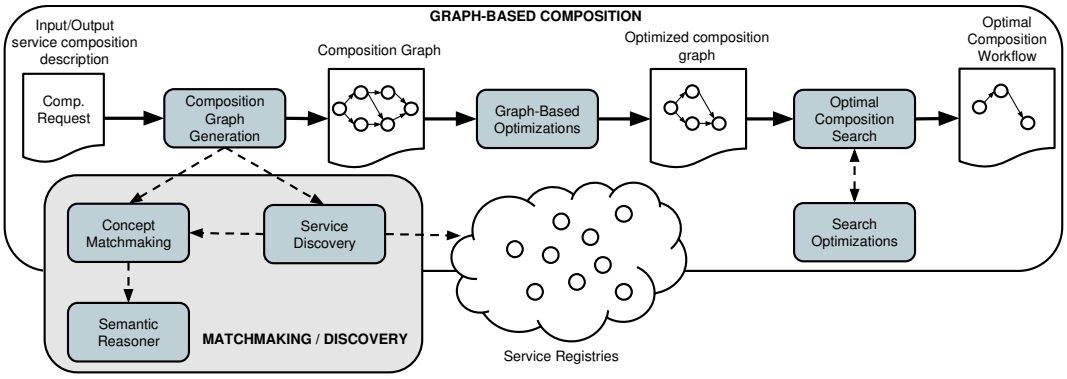

Figure 2. Graph-Based Semantic Service Composition Process

Graph-based approaches constitute a common strategy for tackling the composition problem where nodes in the graph represent services and edges represent input-output matching between them. The kinds of input-output matching that are acceptable (e.g., *Exact*, *Plugin*, etc.) is a configurable aspect, although only *Exact* and *Plugin* matches can ensure direct compatibility, and are therefore the only ones typically contemplated. Figure 2 provides the overview of the approach adopted within the Assisted Composition Engine, which adopts a graph-based approach.

The first step in our approach starts with a composition request that specifies the request of the user in terms of inputs (i.e., the information that the user provides and can be consumed by the composite service) and the outputs expected (i.e., the outputs that are expected to obtain after the execution of the composite service). This information is then used during the composition graph generation phase to recursively build a graph with all potentially relevant services and the semantic relations between their inputs and outputs. This first step thus essentially generates a search space restricted only to those services that could in principle be involved in a suitable composition. In general this graph will contain a significantly reduced search space compared to dealing with all known services and thus its outcome is a reduce problem search space that should yield performance improvements.

In order to find the relevant services, the composition graph phase is interleaved with a service discovery phase whereby, given the set of data types available in the composition graph, we obtain potentially relevant services. Those services are those that could be invoked with the data available and would yield new data. Therefore, to this end, as the graph generation proceeds, the engine contacts all the known service registries in order to find services that could match some of the types of data available within the graph.

The relations between the inputs and outputs of services are computed in the matchmaking phase. During matchmaking, the semantic matching degree between the inputs and outputs of services (see earlier) is calculated using a semantic reasoner. Indeed, the reasoning capabilities of the infrastructure will determine the kinds of ontology representation languages that can be supported. In our case, the reasoning infrastructure is provided by iServe, which is currently able to support OWL Horst thanks to using OWLIM, see D3.1.1.1. Note, however, that if

necessary more expressive languages and reasoners could be used without any significant matchmaking performance impact since reasoning takes place when uploading services to the registry and the results are materialised for fast querying.

Once the matchmaking between inputs and outputs has been performed the service composition graph is generated using the relevant services and the I/O matchmaking information. The resulting graph basically represents all possible service compositions that could be invoked[1] based on the request. In the cases where solutions exist, this graph will contain, among others, all the compositions that provide a suitable solution to the composition problem at hand.

The service composition graph is then optimized by applying different techniques to group and reduce the number of services and relations. This optimization may include for instance the pruning of services that yield more results given the same input data, or the removal of services that were added to the graph but do not contribute towards a solution (e.g., because they are never used afterwards).

Next, a search is performed over the graph to find possible service compositions that fulfil the request. Given that the problem search space is represented as a graph, the search for a service composition is essentially a graph search that starts from the input data provided and should eventually reach the required output data. This phase is informed by specific criteria, e.g., heuristic, that helps disregard potential paths or simply explore first possible branches given their potential contribution. This criteria can for instance be the total cost of a composition, its expected response time, etc. Finally, the possible composition workflows are returned.

2.2 Assisted Composition Engine Architecture

The Assisted Service Composition Engine devised in COMPOSE is a graph-based engine that approaches the problem from an integrated perspective aiming at reusing existing infrastructure, notably discovery engines, and maximizing the performance of the composition. The engine is composed of three fundamental components, namely a Semantic I/O Matchmaking system, a Semantic I/O Service Discovery system, and the Graph-Based Semantic Service Composition system.

It is worth noting that the discovery engine (see D3.1.1.1), already needs to address the first two activities to carry out service discovery, and also provides access to the pool(s) of known services. It is therefore fundamental to adequately enable exploiting the functionality offered by the discovery engine within the composition engine in a way that is convenient both from an algorithmic and a performance perspective. We have therefore devoted significant efforts in this work towards adequately defining the necessary discovery interface and adapting COMPOSE's discovery engine so as to complement the advanced support for service discovery by end-users with support for efficiently discovering and matching services while computing service compositions.

[1] Assuming that we only contemplate Exact and Plugin input-output matches.

Generally, discovery engines try to find suitable services based on a coarse-grained request defined as prototypical service description in terms of an input-output signature and (a set of) functional classification(s). The goal of discovery is finding atomic services that match the request, that is, services that offer the functionality required and match the interface description (i.e., inputs and outputs). This search typically considers the description of the services to match as a whole, that is, the interface and functionality descriptions are treated in a way such that they need to be entirely fulfilled[2]. Additionally, discovery engines typically envisage the interaction with clients as a one-of activity whereby clients are expected to generate sporadically fully-fledged requests for entire services, and response time is not a fundamental concern.

Conversely, though, composition engines do not aim at finding the best candidates for the request for they usually do not exist as atomic services. Instead they try to find suitable combinations of services that satisfy the request. In this scenario, useful services need to be located very fast using partial and incomplete information that is available in each step of the composition search. This in essence requires carrying out discovery at a more fine-grained level whereby even the fact that a service can process or generate one of the semantic data types within a workflow may be a contribution to the final solution. Last but certainly not least, a large number of such finer-grain discovery requests need to be fulfilled in a short time.

In order to facilitate the integration of the discovery and composition tasks, the discovery engine should provide methods to easily discover services that are relevant for parts of the interface available (individual inputs or outputs) instead of focusing solely on total coverage of the interface required. This process has been grounded in two main open source software components as shown in Figure 3. On the one hand, we take COMPOSE's discovery engine iServe, see D3.1.1.1, as starting point for providing service discovery support. On the other hand, we use ComposIT [13], an open source graph-based service composition engine developed by the University of Compostela, as a starting point for service composition. The choice of ComposIT is mainly driven by the following main factors: i) its performance which already highlighted high efficiency (see Section 3.2 for a performance comparison); ii) its availability as open source; and iii) its modular approach to semantic service composition which provides a good starting point for a closer integration with discovery and further functionality envisaged in COMPOSE.

As usual both for discovery and composition engines, though, both components were initially designed to deal with their respective tasks in a holistic manner without contemplating let alone exploiting potential synergies and dependencies between the composition and discovery tasks. In particular, ComposIT included its own basic discovery mechanisms to be run in memory, assuming that all services are available for pre-loading. iServe on the other hand provided a discovery interface with granular access to different discovery techniques, but its interface was thought for one-of service discovery whereby very low response time had not been a fundamental factor.

Based on the generic framework for composition presented earlier, we define the interface that iServe, or any other discovery engine for that matter, should have and provide concrete

[2] Some engines contemplate partial matches among potential results, although this is less common, e.g., [12].

performance requirements in order to enable efficient service compositions. Indeed, this interface is informed by the particular graph-based approach followed by the engine but based on an analysis of state of the art engines we posit that the interface would adequately support a wide range of composition engines.

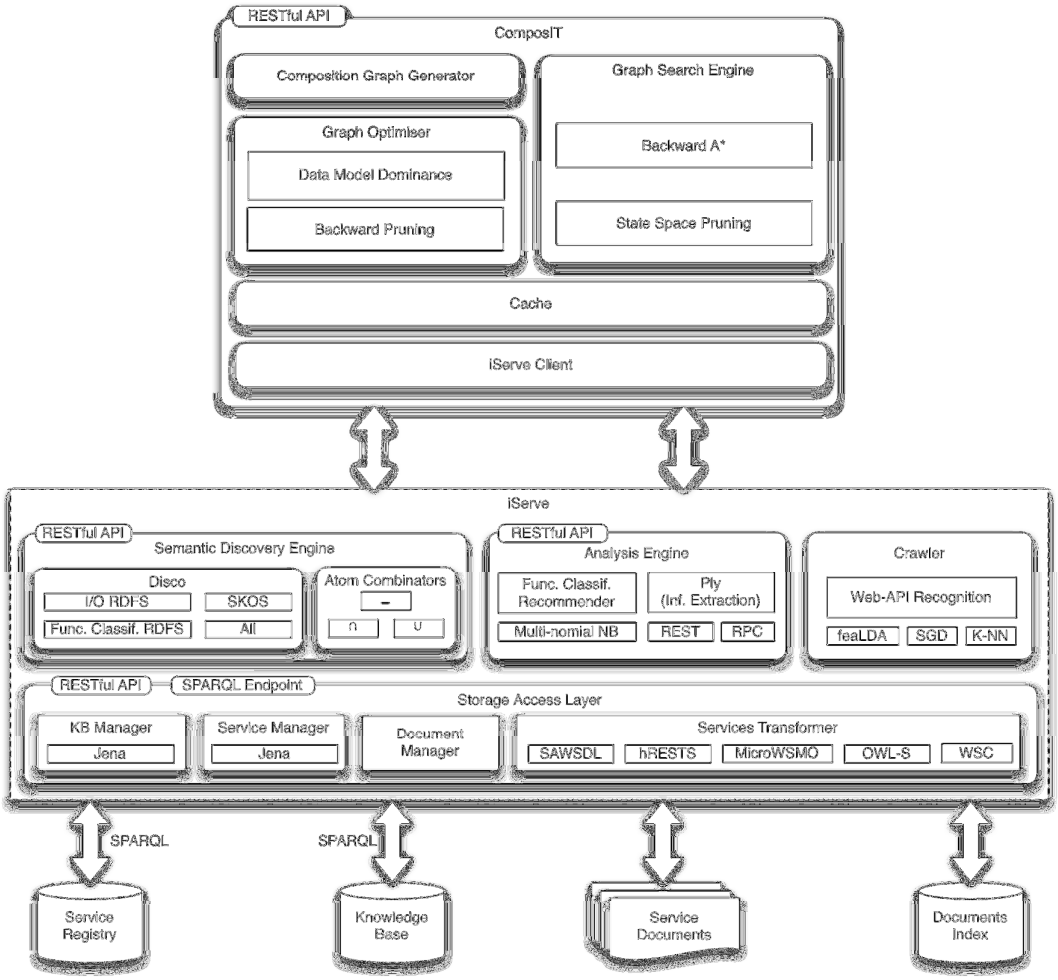

Figure 3. Architecture of the Assisted Composition Engine.

2.3 Software Components

The Composition Engine is made of four main components. We shall describe them briefly in the remainder of this section.

2.3.1 Composition Graph Generator

The Composition Graph Generator is in charge of, given a composition request described in terms of input and output data types, computing a graph with all the semantic relations (e.g., *Exact*, *Plugin*, etc) between all the inputs/outputs of all potentially relevant services known to the registries used. The resulting graph is a Directed Acyclic Graph that captures all potential composite services in a number of layers. In a nutshell, every layer captures the services that can be executed in parallel at that stage of the workflow given the set of available semantic data types. Whereby the available types are basically the aggregation of those available from the start together with those that would be produced by the execution of all the services in the preceding layers of the composition graph. In a nutshell, this graph constitutes the problem search space, which is a subset of the entire space of possible compositions that contains all possible solutions that fulfill the request.

Figure 4 shows an example composition graph of four layers, whereby all the possible service compositions that fulfil a request have been generated. Finding a solution to the composition request would be a matter of finding paths starting from the first layer (the available data) and leading to the last layer (the required data). As can be seen, there are some viable paths, e.g., W1,W2, W4, W8 (where W1 and W2 could run in parallel), and many others that are not viable.

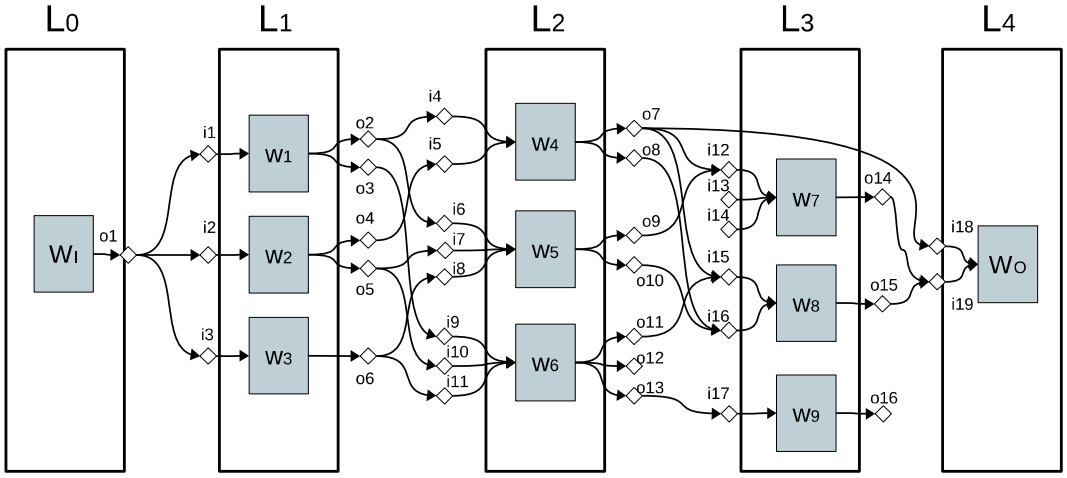

Figure 4. Example of a Composition Graph.

2.3.2 Graph Optimiser

Once the graph is generated, the graph optimiser applies different optimizations to reduce even further the composition graph so as to increase the composition search performance. This part of the composition is independent of the discovery phase. The current version of the composition engine implements two different techniques, namely *backward pruning* and interface dominance, which can independently be applied to reduce the search space.

Backward Pruning

The composition graph is generated forwards starting from the inputs of the request and including at every step all new invocable services given the set of inputs available. As a result there may in the end be services that although they are invocable, they do not contribute to the final solution, i.e., they do not provide any additional output data necessary for the final solution. Backward pruning explores the graph backwards starting from the final layer, and removes all services that do not contribute to the final solution.

Interface Dominance

Interface dominance is another strategy for reducing the search space, which is based on exploiting the fact that there may be services that have an interface that dominates the interface of other candidates. Service A is understood to be interface-dominant over Service B if it is input-dominant and/or output-dominant. Service A is said to be input-dominant over Service B if it has a less demanding input interface, i.e., it requires less inputs to be invoked, and generates at least the same semantic output types. Conversely, Service A is said to be output-dominant over Service B if it has a richer output interface, i.e., it generates further outputs, and requires at most the same semantic inputs types to invoked.

In a nutshell interfaces dominance allows pruning the search space by removing services that although they could be part of a final composition, their contribution to the final solution would be at most as good as that of dominant candidates. It is worth noting that this optimisation only takes into account the interface of the services and would therefore need to be used carefully whenever other aspects, e.g., non-functional parameters, are crucial. Indeed, some viable compositions, although possibly less optimal, could be reached through the pruned branches, so whenever all possible compositions must be obtained, this optimization should not be used.

2.3.3 Graph Search Engine

The Graph Search Engine is in charge of retrieving the desired compositions from the composition graph resulting from the previous steps. In a nutshell, given the Directed Acyclic Graph that captures the possible compositions for a particular request, this component performs a graph search among all possible compositions that satisfy the interface (i.e., inputs and outputs) of the request. This search can be designed to optimise diverse criteria such as the length of the composition (i.e., number of services invoked sequentially), the expected response time, etc.

In the current implementation of the composition engine, this search is performed using a backwards A* pathfinding algorithm. This algorithm, well-known for its performance and accuracy, uses a best-first approach in order to find the least-cost path between an initial node and the goal. At every step the algorithm chooses the next candidate based on a heuristic function that takes into account the cost so far, and the estimated additional cost of choosing this path. In the current implementation the current cost is the number of services which are part of the solution, and the heuristic estimation of the future cost is based on the current layer of the exploration graph. Bearing in mind that we explore the graph backwards this estimation tends to be accurate and can be computed easily.

2.3.4 iServe Client

During the composition phases described earlier, a considerable number of interactions are required between the composition engine and the discovery engine(s) being used. This is notably the case for the construction of the composition graph where many discovery queries need to be issued. The composition engine has been developed in a generic way whereby specific discovery engines may be integrated through dedicated clients. The iServe client is in charge of generating the required queries to COMPOSE's discovery engine.

2.4 Java APIs

The composition engine functionality is currently exposed by means of a Java API. This Java API will eventually be also wrapped as a RESTful API so that applications may make use of the composition support either through direct embedding or remotely via HTTP. Within COMPOSE, the composition engine is expected to be used mainly by the GUI for supporting developers in creating new services and applications. We shall, however, expose the entire API as part of the platform so that this can also be used from IDEs and from third party applications directly.

2.4.1 Composition Engine Interface

Low-level constructor for developers of the composition engine or users wanting full control over the instantiation process.

```
CompositIserveEngineImpl CompositIserveEngineImpl(
      iServeEngine discoveryEngine,
      OperationTranslator operationTranslator,
      InputDiscoverer<URI> inputDiscoverer,
      MatchGraph<URI, LogicConceptMatchType> matchGraph,
      @Assisted
      List<NetworkOptimizer<URI, LogicConceptMatchType>>
optimisations,
      @Assisted
      Integer matchCacheSize)
```

The construction of an engine requires:

- An implementation of *iServeEngine* that allows interacting with iServe. iServe's code can directly be used to this end, see D3.1.1.1.
- An *OperationTranslator* to carry out the transformation between iServe's internal data model and the composition engine's one. A reference implementation is already provided.
- An *InputDiscoverer* which is in charge of performing service/operation discovery driven by the inputs they can consume. A reference implementation is already provided.

- A *MatchGraph* implementation which is in charge of capturing the composition graph. A reference implementation is provided.
- A list of *NetworkOptimisers*. NetworkOptimisers are simply classes that can, given a composition graph, obtain a smaller graph filtered out by some criteria. The engine already provides two implementations: *BackwardMinimizationOptimizer* and *InterfaceDominanceOptimizer*.
- A *matchCacheSize* to reduce the interaction with the discovery engine by caching some results. This parameter is more relevant for the future RESTful-based interface than it is in the embedded integration.

Convenient constructor through simple Factory method for regular users.

```
CompositIserveEngine create(
      List<NetworkOptimizer<URI, LogicConceptMatchType>>
optimisations,
      Integer matchCacheSize)
```

The construction of an engine requires:

- A list of *NetworkOptimisers*. NetworkOptimisers are simply classes that can, given a composition graph, obtain a smaller graph filtered out by some criteria. The engine already provides two implementations: *BackwardMinimizationOptimizer* and *InterfaceDominanceOptimizer*.
- A *matchCacheSize* to reduce the interaction with the discovery engine by caching some results. This parameter is more relevant for the future RESTful-based interface than it is in the embedded integration.

```
Result compose(Set<URI> inputs, Set<URI> outputs)
```

This method triggers the actual computation of compositions given a client request. The request includes:

- *Inputs*, a set of URIs where every URI is the identifier of the semantic type of an input available.
- *Outputs,* a set of URIs where every URI is the identifier of the semantic type of an output required.

2.4.2 Required Discovery Engine Interface

The interface should provide both high-level discovery methods over services and operations, as well as lower level matching methods that can identify the different matches between service/operation inputs and outputs. The former methods are used in order to obtain the nodes that will constitute the graph of potential compositions, whereas the latter methods help generate the edges within that graph. These methods include support for 1:1 and N:N matching in order to support more efficient computation.

For all the following interfaces, MatchResult is a data structure that tracks what was matched to what (by means of URIs of services or operations), and the MatchType (i.e., Exact, Plugin, etc.).

These data structures are used extensively by the discovery engine internally. Similarly, for all the methods, the URIs passed as parameters are the URIs of concepts capturing the semantics of an input or output depending on the actual method. The result of all these methods is a Map mapping the URI of the matched resource (i.e., a service or an operation), and the actual MatchResult (i.e., how it was matched). The MatchResult is important for inspection where as the indexing by URI allows efficient merging of MatchResults by URI in cases where the same resource is matched by different discovery methods. This essentially allows creating complex discoveries by composing on the fly the results of simpler ones.

Required Interface for Service Discovery

```
Map<URI, MatchResult> findServicesConsumingAll(Set<URI> inputTypes)
```

This method finds all the services consuming all the semantic input types given.

```
Map<URI, MatchResult> findServicesConsumingSome(Set<URI> inputTypes)
```

This method finds all the services consuming some the semantic input types given.

```
Map<URI, MatchResult> findServicesProducingAll(Set<URI> outputTypes)
```

This method finds all the services producing all the semantic input types given.

```
Map<URI, MatchResult> findServicesProducingSome(Set<URI> outputTypes)
```

This method finds all the services producing some the semantic input types given.

Required Interface for Operation Discovery

```
Map<URI, MatchResult> findOperationsConsumingAll(Set<URI> inputTypes)
```

This method finds all the operations consuming all the semantic input types given.

```
Map<URI, MatchResult> findOperationsConsumingSome(Set<URI> inputTypes)
```

This method finds all the operations consuming some the semantic input types given.

```
Map<URI, MatchResult> findOperationsProducingAll(Set<URI> outputTypes)
```

This method finds all the operations producing all the semantic input types given.

```
Map<URI, MatchResult> findOperationsProducingSome(Set<URI>
outputTypes)
```

This method finds all the operations producing some the semantic input types given.

Required Interface for Matchmaking

```
MatchResult match(URI origin, URI destination)
```

This method performs a one-to-one matching between two resources (e.g., two services) and returns the actual matching which could be Exact, Plugin, Subsume or Fail.

```
Table<URI, URI, MatchResult> match(Set<URI> origins, Set<URI>
destinations)
```

This method performs n-to-m matching between n origin resources (e.g., services) and m destination resources. The result is a Table (i.e., a Matrix) capturing all the matches.

```
Map<URI, MatchResult> listMatchesOfType(URI origin, MatchType type)
```

This method lists all the resources that match the given origin with the given type. Whereby the type can be one of Exact, Plugin, Subsume and Fail.

```
Table<URI, URI, MatchResult> listMatchesOfType(Set<URI> origins,
MatchType type)
```

This method lists all the resources that match the given origins with the given type. Whereby the type can be one of Exact, Plugin, Subsume and Fail. The result is a matrix of matches.

```
Map<URI, MatchResult> listMatchesAtLeastOfType(URI origin, MatchType
minType)
```

This method lists all the resources that match the given origin with at least the given type. Whereby the type can be one of Exact, Plugin, Subsume and Fail.

```
Table<URI, URI, MatchResult> listMatchesAtLeastOfType(Set<URI>
origins, MatchType minType)
```

This method lists all the resources that match the given origins with at least the given type. Whereby the type can be one of Exact, Plugin, Subsume and Fail.

3 Performance Evaluation

In order to evaluate the composition engine we used existing evaluation datasets and we compared the performance obtained to that of state-of-the-art semantic Web services composition engines. Concretely, we used for our evaluations the datasets from the WSC'08[3] contest, and we used as baseline for comparison the semantic planners PORSCE-II [6] and OWLS-Xplan 2.0 [15]. These engines were notably selected due to their availability as open source tools.

It is worth mentioning that the baseline engines are OWL-S composition engines. We thus transformed the entire WSC'08 dataset into OWL-S so as to support comparing the results. Additionally, as explained earlier and as opposed to the baseline engines, at this stage we disregard preconditions and effects due to the fact that they are seldom available. Thus, this first prototype, could yield composition results that do not honour preconditions and effects should these be available.

[3] http://www.it-weise.de/documents/files/BBKBWJ2008WSC08CTWSC_flyer.pdf

3.1 Datasets

In order to carry out the experiments we obtained (or derived) from WSC'08 3 families of datasets. First, we used the datasets provided by WSC'08 directly which require exploiting the semantics of concepts from a reference taxonomy provided by the dataset, in order to obtain viable compositions (see Table 1). Every dataset, includes a number of services, a reference taxonomy of differing size (in terms of concepts included). For every dataset reference queries are provided for which the solutions are known in advance. We include in Table 1 both the number of services in the solution as well as the length of the workflow, i.e., the largest path of services that need to be invoked sequentially.

Second, we scaled down these datasets to obtain smaller scale ones that could also be handled by engines that are not prepared for large-scale problems (see Table 2). Table 2 includes details about the average number of inputs and outputs per service. The reason these details are tracked is due to the impact different ratios have on the exploration of the problem space. For instance, scenarios with a high number of average outputs compared to the inputs, lead to a very fast increase in the paths to explore.

Finally, we also transformed these smaller datasets to obtain a version in which any additional facts are pre-derived and embedded within the dataset so that engines not directly exploiting the semantics of ontologies (e.g., those not benefiting subsumption reasoning) could still work (see Table 3). This 3rd family of datasets was generated in order to better determine the impact of subsumption reasoning within the engines tested. Each of the datasets was converted into MSM as necessary for COMPOSE, and into OWL-S as necessary by the baseline engines.

Table 1. WSC'08 Datasets

Dataset	#Services	#Concepts	#Services Solution	Solution Length
WSC'08 01	158	1540	10	3
WSC'08 02	558	1565	5	3
WSC'08 03	604	3089	40	23
WSC'08 04	1041	3135	10	5
WSC'08 05	1090	3067	20	8
WSC'08 06	2198	12468	40	9
WSC'08 07	4113	3075	20	12
WSC'08 08	8119	12337	30	20

Table 2. Semantic Match Datasets

Dataset	#Services	#Services Solution	#Depth	Avg. In / Out	#Init concepts	#Goal concepts
Exact-Match 01	2	2	2	1.0 / 1.0	1	2
Exact-Match 02	3	3	2	3.0 / 20.5	3	40
Exact-Match 03	10	10	3	4.03 / 16.53	3	100
Exact-Match 04	5	5	3	4.25 / 21.05	10	77
Exact-Match 05	40	40	23	5.07 / 14.20	10	407
Exact-Match 06	10	10	5	5.47 / 24.09	28	157
Exact-Match 07	20	20	8	3.03 / 26.39	8	261
Exact-Match 08	35	35	14	3.10 / 23.21	42	507
Exact-Match 09	20	20	12	3.07 / 24.71	33	219
Exact-Match 10	30	30	20	3.25 / 33.81	23	473

Table 3. Exact Match Datasets

Dataset	#Services	#Services Solution	#Depth	Avg. In / Out	#Init concepts	#Goal concepts
Exact-Match 01	2	2	2	1.0 / 1.0	1	2
Exact-Match 02	3	3	2	3.0 / 20.5	3	40
Exact-Match 03	10	10	3	4.03 / 16.53	3	100
Exact-Match 04	5	5	3	4.25 / 21.05	10	77
Exact-Match 05	40	40	23	5.07 / 14.20	10	407
Exact-Match 06	10	10	5	5.47 / 24.09	28	157
Exact-Match 07	20	20	8	3.03 / 26.39	8	261

Exact-Match 08	35	35	14	3.10 / 23.21	42	507
Exact-Match 09	20	20	12	3.07 / 24.71	33	219
Exact-Match 10	30	30	20	3.25 / 33.81	23	473

3.2 Experiments

All the experiments were performed using a virtualized Windows XP 32-bit with VirtualBox 4.1.22 (2 GB, 1 core).

3.2.1 Baseline Evaluation with Other Composition Engines

The first batch of experiments, see Table 4, was carried out on the basis of the simplest test-cases, the exact matches datasets. In these tests all engines are pre-loaded with an in-memory copy of the dataset. Additionally, PORSCE is configured so as to avoid exploring the subsumption hierarchy. The results show that for small datasets the performance of XPlan and ComposIT is in the same order of magnitude even though ComposIT performs in general better. PORSCE on the other hand has a response time an order of magnitude slower. What can also be observed is that ComposIT identifies situations where service invocation can be parallelised, whereas the other engines always provide sequential solutions where the length of the composition is equal to the number of services. In fact, as it can be observed, for XPlan and PORSCE the length of the workflow is always equal to the number of services whereas ComposIT returns workflows where the length is smaller thanks to its ability to identify parallelizable invocations.

For larger datasets, the situation is fairly similar except for the fact that for all datasets with 20 or more services, XPlan crashed. PORSCE could handle larger scale datasets but did also crash in several tests. Thus, although both engines are able to handle preconditions and effects as opposed to ComposIT, both baseline engines exhibited low scalability even when no subsumption reasoning or precondition and effects evaluation is necessary. It can also be observed that engines although PORSCE found results some times the composition where unnecessarily complex since the number of services in the solution is way larger than those found by ComposIT. Indeed, should the compositions obtained by PORSCE be used for execution at a larger stage, this would have a negative impact on cost and complexity.

Table 4. Exact Matching Experiments

Dataset	Algorithm	Solution		Execution Time (ms)		
		#Serv.	#Length	Min	Avg	SD +/-

Exact-Matching 01	ComposIT	2	2	145.07	169.71	38.57
	Xplan 2.0	2	2	305.37	329.67	24.29
	PORSCE (0)	2	2	3478.53	3586.47	98.6
Exact-Matching 02	ComposIT	3	2	196.24	219.6	31.28
	Xplan 2.0	3	3	391.3	439.37	42.17
	PORSCE (0)	9	9	5206.54	5372.56	188.49
Exact-Matching 03	ComposIT	10	3	330.27	373.54	33.19
	Xplan 2.0	10	10	595.58	658.71	89.38
	PORSCE (0)	21	21	14779.84	15628.1	686.69
Exact-Matching 04	ComposIT	5	3	316.42	338.19	29.13
	Xplan 2.0	-	-	-	-	-
	PORSCE (0)	13	13	10595.46	10785.57	122.58
Exact-Matching 05	ComposIT	40	23	6669.89	6788.67	185.2
	Xplan 2.0	-	-	-	-	-
	PORSCE (0)	-	-	-	-	-
Exact-Matching 06	ComposIT	10	5	463.92	505.22	57.2
	Xplan 2.0	-	-	-	-	-
	PORSCE (0)	28	28	23880,96	24217.28	332.65
Exact-Matching 07	ComposIT	20	8	895.89	983.54	79.27
	Xplan 2.0	-	-			
	PORSCE (0)	52	52	35577,92	36810.72	1043.49

Exact-Matching 08	ComposIT	35	14	5859.83	6145.85	189.58
	Xplan 2.0	-	-	-	-	-
	PORSCE (0)	-	-	-	-	-
Exact-Matching 09	ComposIT	20	12	1065.36	1086.53	14.09
	Xplan 2.0	-	-	-	-	-
	PORSCE (0)	-	-	-	-	-
Exact-Matching 10	ComposIT	30	20	5236.27	5422.38	195.88
	Xplan 2.0	-	-	-	-	-
	PORSCE (0)	-	-	-	-	-

The second batch of experiments, see Table 5, was performed using the semantic match datasets, i.e., the scaled down version of the original WSC'08 dataset[4], which requires subsumption reasoning. For this experiment we have to fine tune PORSCE for one can configure how far the engine follows the subsumption hierarchy. We carried out experiments and saw that setting this value to 2 allowed the engine to cover 30% of the problems with good performance, and the maximum success rate, i.e., 60% of the problems, were solved with the parameter set to 6. We report all results for both cases so as to adequately account for the performance of PORSCE.

The results highlight that again ComposIT has a performance which is an order of magnitude better. Again there is a relatively low threshold of scalability above which PORSCE was unable to find solutions. This threshold is indeed lower when the engine is configured to follow the subsumption hierarchy up to 6 levels. The reader may notice that no results are presented for XPlan 2.0. The reason for this is that, as many classical state-based action planning algorithms, OWL-S XPlan 2.0 does not perform subsumption reasoning unless the subsumption hierarchy is replicated as preconditions and effects. That is, out of the box OWL-S XPlan 2.0 does not deal with ontologies, it deals only with PDDL rules.

[4] The number of services contemplated was considerably lower to ensure that the baseline engines obtained results.

Table 5. Semantic Match Experiments

Dataset	Algorithm	Solution		Execution Time (ms)		
		#Serv.	#Length	Min	Avg	SD +/-
Semantic-Match. 01	ComposIT	2	2	92.46	220.12	125.32
	PORSCE (2)	2	2	9767.84	10501.26	1260.13
	PORSCE (6)	2	2	21941	23088.75	1378.03
Semantic-Match. 02	ComposIT	3	2	87.49	221.57	121.35
	PORSCE (2)	3	3	13685.42	14414.49	449.48
	PORSCE (6)	3	3	32920.82	33484.18	491.57
Semantic-Match. 03	ComposIT	10	3	104.14	243.8	85.79
	PORSCE (2)	11	11	45359.04	47694.39	2076.83
	PORSCE (6)	11	11	107141.89	111926.2	6893.53
Semantic-Match. 04	ComposIT	5	3	97.21	162.47	84.48
	PORSCE (2)	-	-	-	-	-
	PORSCE (6)	6	6	53566.22	57363	4175.13
Semantic-Match. 05	ComposIT	40	23	272.57	523.31	224.32
	PORSCE (2)	-	-	-	-	-
	PORSCE (6)	-	-	-	-	-
Semantic-Match. 06	ComposIT	10	5	96.48	191.31	132.1
	PORSCE (2)	-	-	-	-	-
	PORSCE (6)	14	14	112889.12	118878.62	8806.96
Semantic-	ComposIT	20	8	132.44	277.83	115.8

Match. 07						
	PORSCE (2)	-	-	-	-	-
	PORSCE (6)	35	35	228925.4	244652.12	13963.95
Semantic-Match. 08	ComposIT	35	14	338.21	680.53	202.71
	PORSCE (2)	-	-	-	-	-
	PORSCE (6)	-	-	-	-	-
Semantic-Match. 09	ComposIT	20	12	151.83	351.61	219.36
	PORSCE (2)	-	-	-	-	-
	PORSCE (6)	-	-	-	-	-
Semantic-Match. 10	ComposIT	30	20	210.08	325.16	154.28
	PORSCE (2)	-	-	-	-	-
	PORSCE (6)	-	-	-	-	-

3.2.2 Performance Evaluation of the ComposIT-iServe Integration

A final set of experiments was carried out over the WSC'08 datasets. In this case, the experiments were only performed with ComposIT since the other engines could not tackle problems of this size. In this set of experiments we carried out a more detailed analysis to better understand the performance obtained, where most execution time is spent, etc. These experiments highlighted the following main conclusions that have driven this work to a large extent. In carrying out these experiments, we notably explored the integration between ComposIT and iServe. These experiments were used to analyse the impact that discovery performance has on composition, but also these tests were instrumental in guiding the development of better performing discovery infrastructure that could lead to good performance composition results.

First and foremost, these experiments made pretty evident that the typical discovery interfaces offered by engines (an initially implemented in iServe) are not suitable for supporting composition. Initial integrations showed a granularity mismatch between the interface offered by the discovery engine and what was required by the composition engine. Secondly, it showed that regardless of the granularity issue, the performance demand for ensuring that compositions

are computed with a reasonable performance require using more advanced infrastructure in the discovery engine.

As part of this work, iServe was extended to implement a new API (see Section 2.4.2) which can better support the kinds of discovery requests issued by a composition engine. This new API was implemented using 3 different configurations:

- **SPARQL D/M** is based on the original implementation of iServe, which relies directly and solely on issuing SPARQL queries both for service discovery and for figuring out the type of match between 2 concepts.
- **Index. D/SPARQL+Cache M** is based on *SPARQL concept matching* with a local cache and an indexed I/O Discovery, where the I/O discovery mechanism provided by iServe is pre-computed and indexed beforehand, and a local cache memory is used to save frequent calls to the matchmaking system.
- **Full Indexed D/M**, which corresponds with the optimal configuration using full indexed discovery and matchmaking mechanisms that are pre-computed using the information stored in the semantic registries to avoid the use of *SPARQL* queries during the composition time.

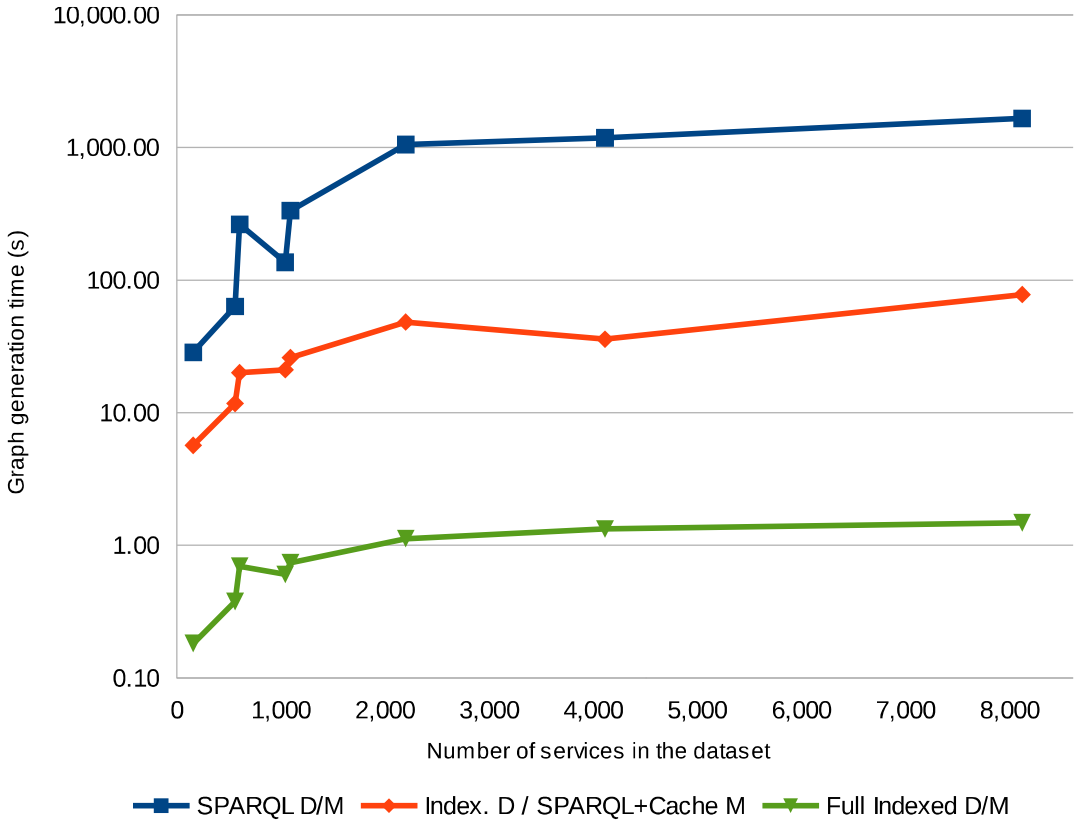

Figure 5. Performance Evaluation of Different Discovery Implementations in iServe

Not surprisingly our experiments, shown in Figure 5, evidence the important impact that discovery has on the overall performance of composition. Most importantly, though, they highlight that discovery solutions based directly on querying the backend (e.g., using SPARQL in this case) which are typically acceptable for one-of discovery requests, are simply not viable for solving composition problems. The performance quickly degrades and leads to response times of more than 1,000 seconds (approx. 17 mins) for registries with more than 2,000 services.

An initial performance improvement was developed which uses an index of the input-output matches between all services/operations. This index provides a major reduction in the execution performance which enables computing compositions in less than 100 seconds for registries with 8,000 services.

Despite the improvements, this intermediate index still does not allow achieving response times that could be acceptable for users in live settings. A final improvement was developed based on the indexing of the matches between all concepts in addition to the input-output matching index. This mechanism, which we refer to as *full indexed*, allowed us to compute compositions with registries of 8,000 services within a bit more than a second in a basic machine.

Indeed, performance improvements come at the price of additional memory requirements. Still, the index being based solely on maps between URIs, the memory requirements are minimised. We nonetheless envisage carrying out further tests using distributed hybrid infrastructures for the index, whereby only a subset of the index is kept in memory.

4 Future directions

The next prototype will benefit additionally from further solutions developed within the project. These solutions will notably help the engine devise further general purpose optimisations and search heuristics, as well as they will enable carrying out more complex checks over the resulting compositions, notably from a security perspective.

4.1 Exploiting Non-Functional Properties

The current version of the composition engine taps directly into the input/output discovery functionality. There is, however, ongoing additional work on a recommender engine, see D3.1.2.1, and on the capturing of general non-functional properties about services, see WP3.2. The final version of the prototype will look into incorporating these properties and the ranked results derived from them, in order to guide the exploration of potential solutions, e.g., by selecting most promising services first, and by enabling the use of other optimality criteria beyond that of the shortest path.

Indeed, however, rapid response time will remain a must and we shall therefore look into developing the appropriate indexes and adaptations in a way such that high performance is retained. Using other heuristics for guiding the search space exploration will most likely be taken into account during the overall composition process. However, the application of diverse optimisation criteria will require more careful analysis of possible integrations ranging from interleaved checks (possibly in combination with the new heuristics) to post-composition ranking.

4.2 Post-analysis for Security Compliance

As introduced in deliverable D5.1.1, services carry pre- and post-conditions which may determine access control rules. Further, they are annotated with flow rules. Both rule-sets form the security policy for a service. These rules are not considered in the first phase of the composition generation to reduce its complexity and keep the delay within an acceptable time frame. Nevertheless, future work will include a post-filtering step which checks the generated compositions for their compliance with the security policies. This will prevent the user from running services that cannot be executed because the COMPOSE runtime prevents this.

For this purpose, WP5 will use a graph representation of the composition enriched with the rules above. On this graph combined with additional information, such as the user who is composing and for whom, we will run a policy check. This check will be twofold.

First, a policy compliance check will be run by the analysis. Policies validate whether particular principals in the platform are allowed to use a particular service or service composition. Here the service is considered as one single unit. Based on the policies encountered during this analysis we also derive a policy for the analysed composition. This information is stored as meta-data for the services in the security component (see D5.1.1).

The second check applied in this analysis will validate flow compliance. It will use the automatically and semi-automatically generated contracts of a service (see Deliverable 5.1.1). It verifies whether pre- and post-conditions of the dataflows generated by the composition engine are compliant. During this compliance check, we also derive the appropriate rules, for the analysed composition. This additional meta-data about service composition will also be stored by the security component.

In the final year of this project we may also investigate how unmet conditions can be satisfied by complementing service compositions with appropriate security services (partial outcome of Task 5.5). Based on the meta-data of a composition, we will attempt to discover security services which deliver security primitives which help to render a composition to be compliant with the security requirements.

5 References

[1] M. P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann, "Service-Oriented Computing: State of the Art and Research Challenges," *Computer*, vol. 40, no. 11, pp. 38–45, 2007.

[2] S. Dustdar and W. Schreiner, "A survey on web services composition," *Int. J. Web Grid Serv.*, vol. 1, no. 1, pp. 1–30, 2005.

[3] J. Rao and X. Su, "A Survey of Automated Web Service Composition Methods," in *Semantic Web Services and Web Process Composition*, vol. 3387, no. 5, Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 43–54.

[4] C. Pedrinaci, A. Sheth, and J. Domingue, "Semantic Web Services," in *Handbook of Semantic Web Technologies*, 1st ed., vol. 2, no. 22, J. Domingue, F. Dieter, and J. A. Hendler, Eds. Springer, 2011.

[5] M. Klusch and A. Gerber, "Evaluation of Service Composition Planning with OWLS-XPlan," presented at the Web Intelligence and Intelligent Agent Technology Workshops, 2006. WI-IAT 2006 Workshops. 2006 IEEE/WIC/ACM International Conference on, 2006, pp. 117–120.

[6] O. Hatzi, D. Vrakas, M. Nikolaidou, N. Bassiliades, D. Anagnostopoulos, and I. Vlahavas, "An Integrated Approach to Automated Semantic Web Service Composition through Planning," *Services Computing, IEEE Transactions on*, no. 99, p. 1, 2011.

[7] E. Sirin, B. Parsia, D. Wu, J. Hendler, and D. Nau, "HTN planning for Web Service composition using SHOP2," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 1, no. 4, pp. 377–396, Oct. 2004.

[8] K. Sycara, M. Paolucci, A. Ankolekar, and N. Srinivasan, "Automated discovery, interaction and composition of Semantic Web services," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 1, no. 1, pp. 27–46, 2003.

[9] P. Traverso and M. Pistore, "Automated Composition of Semantic Web Services into Executable Processes," International Semantic Web Conference, 2004, pp. 380–394.

[10] C. Pedrinaci and J. Domingue, "Toward the Next Wave of Services: Linked Services for the Web of Data," *Journal of Universal Computer Science*, vol. 16, no. 13, pp. 1694–1719, 2010.

[11] M. Paolucci, T. Kawamura, T. Payne, and K. Sycara, "Semantic matching of web services capabilities," *The Semantic Web—ISWC 2002*, pp. 333–347, 2002.

[12] M. Stollberg, M. Hepp, and J. Hoffmann, "A Caching Mechanism for Semantic Web Service Discovery," International Semantic Web Conference, 2007, pp. 477–490.

[13] P. Rodriguez-Mier, M. Mucientes, J. C. Vidal, and M. Lama, "An Optimal and Complete Algorithm for Automatic Web Service Composition," *Int J Web Serv Res*, vol. 9, no. 2, pp. 1–20.

[14] W. van der Aalst, A. ter Hofstede, B. Kiepuszewski, and A. Barros, "Workflow Patterns," *Distributed and Parallel Databases*, vol. 14, no. 1, pp. 5–51, 2003.

[15] M. Klusch and A. Gerber, "Semantic web service composition planning with owls-xplan," presented at the Proceedings of the AAAI Fall Symposium, 2005.

Appendix A. Obtaining the Software

This appendix describes the steps taken in order to build a composition engine from the publicly available distribution. The composition engine requires an accessible iServe engine that will be used to perform service discovery.

Assumptions

- Other packages we assume are installed on your system:
  - git
  - maven
  - Sun JDK 1.6 is installed on the system

Obtaining ComposIT

```
git clone https://github.com/citiususc/composit.git
mvn -DskipTests=true install
```

Once the component has been successfully installed for the first time you may want to run the tests at installation time.

```
mvn install
```

Obtaining ComposIT-iServe

```
git clone  https://github.com/compose-eu/composit-iserve.git
mvn -DskipTests=true install
```

Once the component has been successfully installed for the first time you may want to run the tests at installation time.

```
mvn install
```

Appendix B. Using the Composition Engine

This appendix describes how to use the composition engine within an application using the Java API. This assumes that you have managed to compile and install in your local maven repository the engine, see Appendix A.

Adding the Dependency

```
<dependency>
      <artifactId> composit-iserve</artifactId>
      <groupId> uk.ac.open.kmi</groupId>
      <version> 1.0-SNAPSHOT</version>
</dependency>
```

Using it within an Application

```
// Create the composition engine
CompositIserveEngine engine = CompositionEngineFactory.createEngine();

// Create the input/output signature of the problem
Set<URI> inputs = …
Set<URI> outputs = …

// Invoke the engine
ServiceMatchNetwork<URI, LogicConceptMatchType> result =
      engine.compose(inputs, outputs);
```