A Service Composition Construct to Support Iterative Development

Roy Grønmo[1], Michael C. Jaeger[2], and Andreas Wombacher[3]

[1] SINTEF, P.O.Box 124 Blindern, N-0314 Oslo, Norway
Roy.Gronmo@sintef.no
[2] Technische Universität Berlin, FG FLP, Sek. FR6-10, Franklinstrasse 28/29, D-10587 Berlin, Germany
mcj@cs.tu-berlin.de
[3] School of Computer and Communication Sciences, Ecole Polytechnique Federale de Lausanne (EPFL), CH-1015 Lausanne, Switzerland
andreas.wombacher@epfl.ch

Abstract. Development of composed services requires a continues adaptation of the composed service to the changing environment of offered services. Services may no longer be available or may change performance characteristics, price, or quality of service criteria after they have been selected and used in a composition. The replacement of such a service requires a good understanding why this service got selected in the first place. This is hard to accomplish as it is known from software maintenance. Therefore we propose an approach where the conceptual task implemented by a selected service as well as the relationship between task and selected service is explicated and maintained during the complete life cycle of a composed service. This covers the design of the composition, derivation of service search criteria, and the execution of the composed service. The approach has been validated by an implementation in the Service Composition Studio (SERCS) supporting the iterative development of composed services.

1 Introduction

The vision of the service-oriented architecture (SOA) is that there are numerous available services that can be reused by other parties when developing new services. SOA involves searchable registries, such as UDDI for Web services, and technological infrastructure regarding textual descriptions in XML for bindings and protocols. Most attention has been given to Web services, but now also grid and other service types are proposed to be part of the infrastructure. The goal is that services may be easily found, that the services may be reused and composed into new service compositions, and that the binding and execution of these services work seamlessly. Numerous composition languages and tools have been proposed (eg. BPEL [13], OWL-S [5]) to aid the user when building service compositions. The important question which we will try to answer in this paper is: *What should be the main characteristics of a composition language in order to support the iterative service composition development?* When we investigate this question we assume to some extent that the SOA visions have been properly addressed and that there are lots of available, searchable services, possibly with associated Quality-of-Service (QoS) offerings and semantic information.

We define a composition model to be a representation of a larger task into smaller, more basic tasks. The decomposing is done to break a complex task into smaller, more manageable tasks for which we hope to find existing services. If existing services are found for each particular task, then we may solve the large task by calling other, already defined services in the right order. In general, there may be competing services from different providers and with different QoS offerings, but with the same functionality. Thus, these services are alternative candidates for the same task in our composition model.

It is absolutely crucial that the service composition development is iterative since there are a number of factors which might change over time. The pool of available services will change continuously. Some services will be withdrawn, others will be introduced, and the available services may also change their QoS even though their functionality remains intact. Some services may become temporarily or permanently unavailable. Even if there are established contracts with the service providers to ensure that a specific service is available and delivers the QoS as promised, there may be competing services from other vendors with lower price and better QoS offerings available since the time we previously searched for services.

An example for such fast changing service offerings can be observed in emergency situations. Here the situation changes fast and emergency teams request context-specific services. Thus, the teams iteratively have to adapt the composed service depending on the changing situation and context. In emergency situations a lot of resources of different locations and organizations are involved to handle the situation. An example of such a situation has been the Oder flood in Germany 1997. Certain parts of Germany along the river Oder were flooded by the river and emergency teams from all over Germany were sent to support the local authorities in fighting the flood [14]. A lot of resources were needed to secure cities, houses, and embankments along the river. In particular, several ten thousands of people have been involved in this situation. The emergency teams were sent to different locations to help, and they were coordinated by a hierarchic crisis management.

The emergency teams had to navigate to the different locations in an unknown area with the constrained infrastructure imposed by the flood. Each team offers a composed service as help to the crisis management, and in case the offer gets rejected they finish their operation. Otherwise, they are assigned a location for their operation and have to find their way to that location. To illustrate the benefits of our approach in such a scenario, a *determineBestRoute* service is investigated to find the best route to the assigned location. The used *determineBestRoute* service is applied iteratively, and dependent on the assigned location and the status of the flood, different service providers should be selected. In particular, the emergency team already working at the assigned location should provide the route service, because they are most familiar with the local situation. We focus on the *determineBestRoute* service, and use it as a running example within this paper.

This paper is organized as follows; Section 2 gives a motivation for the paper by explaining how existing approaches fail to provide sufficient support for iterative service composition development; Section 3 introduces our contribution, which is a composition construct and its application within a composition language; Section 4 shows how the composition construct enables search and discovery; Section 5 shows how the composition construct enables execution; Section 6 shortly describes the implementation in the SERCS tool; Section 7 discusses our approach; and Section 8 summarizes with conclusions.

2 Related Work

This section describes some of the proposed composition languages and tools, which we have placed in three main groups. We focus on how they support the iterative development, and why they are not sufficient for iterative development.

First we define two key concepts, task and service, where one or both are present (perhaps with different terms) in any service composition language. A task (also commonly called goal) represents a requirement specification for what we want to accomplish. When the tasks are sufficiently defined, they may be used to search for existing services. A task will typically contain a syntactic interface represented by an operation name, input and output parameters. We anticipate also the description of QoS requirements (throughput, availability or the time delay etc.) and of the semantics of the service elements. The semantic information may include an ontology reference for the inputs and outputs, classification specifications of the service operation, preconditions and postconditions. A service, on the other hand, will contain enough information so that an execution can bind to a unique service and invoke it. If we look at Web services as an example, then the four values of WSDL file, service name, port name and operation name will be enough information to call the Web service. Implicitly, then the information of input and output parameters are also given through the unique operation inside the WSDL file. The necessary binding information will vary between service types.

Existing languages and tools can be placed in three groups related to the task and service concepts: pure task-based, evolving from task-based to service-based, and pure service-based. In the following we investigate these groups by identifying languages or tools that belong to each group:

– **Pure task-based.** Pure task-based approaches provide only task constructs as editable constructs to the user. These approaches may have an equivalent to the service construct, but the services will be automatically selected and executed, and the user cannot operate at the service level. Peer's [11] PDDL tool and Ponnekantis' SWORD tool [12] represent two approaches for expressing overall composition tasks (termed goal and rule), and to automatically generate an executable service composition. Peer uses AI planning techniques on goals defined in PDDL, while SWORD relies on a knowledge-base of axioms for each Web service, and a rule-based expert system to generate the executable composition. We fear that the approaches will not scale to handle the large number of available services in the open SOA environment, especially since they deal with automatic decomposition. Tsalgatidou et al. suggest the USQL language [16] to define requirements for single tasks in isolation. Pure task-based approaches without automatic transition to execution, must always be combined with a service-based execution approach in order to sufficiently support service composition.

– **Evolving from task-based to service-based.** The approaches in this group start out with modeling task-based composition that evolves to a service-based composition when suitable services are identified for each task. The services typically replace the tasks, and the tasks are no longer in active use at the later development stage. Traverso and Pistore [15] present an approach to automatically transform OWL-S [5] process models into executable BPEL documents. An OWL-S process model represents a task-based composition model for which Traverso and Pistore automatically find perfect matches among a set of Web services (theoretically a global registry of services). Perfect matches are then used to build a BPEL document which represents a service-based composition model. Agarwal et al. [1] transform an abstract BPEL document with service types (corresponding to tasks) into a concrete BPEL document with bindings to service instances (corresponding to services). Agarwal et al. allow for the process to be non-automatic with human intervention to select appropriate services, negotiate service-level agreement etc., and to manually define necessary transformations in order to use services. Cardoso and Sheth [4] present a workflow language implemented in the METEOR tool where a service template is a task and a service object is a service according to our definition. The service template is first used to search for service objects, then the developer selects a single service object to replace the service template.

  All the approaches in this group typically start with an original composition graph consisting of nodes representing tasks. The graph is then transformed into a graph with service nodes, where all the task nodes are replaced by service nodes. The problem with this approach is that we have lost all the original information about the tasks which were designed to search for services. The service nodes lack the QoS requirements, and may also have less generalized input and output parameters as well as possibly lacking the appropriate link to semantic definitions. Thus, we are not able to sufficiently repeat the search for services after some period of time. The iterative development in a continuously changing Web environment is not supported properly.

– **Pure service-based.** Pure service-based approaches provide only service constructs and have no equivalent to the task construct. The languages in this group are focused on making executable service compositions. The pure service-based group includes the graphical alternatives BPMN [3], JOpera [10], KEPLER [2] and the textual alternatives BPEL [13] and USCL [10]. Pure service-based approaches provide no help to define tasks and provide limited or no help to search for services, which means that the iterative development is not sufficiently supported.

The first and last approaches may also be combined so that we use one tool for maintaining a pure task-based composition and another tool for pure service-based composition. The problem with this approach is that we now have two compositions that are identical with respect to control flow and number of nodes. This implies a need to maintain the control flow in two places, one for each of the two compositions. Furthermore the relationship between a task and a service are only implicitly defined by their positions in the two different compositions, such as when we have two equivalent graph structures. When the task-based graph is used to perform a service search, the results will have to be manually registered into the corresponding service-based graph. The solution is likely to be error-prone and this is not satisfactory.

All the existing tools can be placed in one of the three groups above which all have their limitations. These limitations lead to requirements for a service composition language in order to sufficiently handle the iterative development:

– **Service discovery.** The composition can be used as a basis for search and discovery of services to fulfill specific tasks. This means that the tasks need to be associated with QoS requirements and semantic annotation.
– **Service execution.** The composition can be used as a basis for execution, that means calling the part services in the correct order. The services need to be associated with data and control flow including necessary data transformations.
– **Composition using a single structure.** The composition can be represented as a single structure to avoid the error-prone maintenance issue of maintaining several structures or graphs with duplication of control flow or other information.

3 The Approach

Our contribution is based around a new service composition construct, which we call the task-service construct. We illustrate the usefulness of the construct by introducing a graphical composition language with task-service nodes as the basic building block.

3.1 The Task-Service Construct

The task-service construct is used to bridge the task composition part with the service composition part. The task-service construct can be viewed as a composite node containing one task part with zero or more services. The task part can be viewed as a requirement specification, and the services represent actual matches to the task specification. Although there are many differences in the information content of a task and a service (Section 2), we also see that the syntactic information of inputs, outputs and operation name shall be represented for both a task and a service. Still we register this information as separate objects. The reason is that we allow for matching services that are not perfect matches. There may for instance be minor differences with respect to the syntactic way of representing the input and output. Conceptually we may treat such services as matches, which we expect will give more alternative services for the composition developer. The composition developer may still choose to only allow perfect matches today, or in the future especially if the semantically described services become widespread. We will discuss how to handle the mismatching part later by using data transformations. A metamodel of the task-service construct is depicted in the left hand side of Figure 1.

Notice that we allow to register multiple services as matches of the task specification. There may be many competing services providing the same service with different (or similar) QoS offerings. Having many alternatives is particularly useful due to varying service performance and availability over time. This will make the service composition less dependent on single services, as alternative services may be used in place of temporarily unavailable ones. The control flow behavior of the services associated to a task is related to the discriminator pattern identified by Aalst et al. [17].

Considering the emergency scenario in Section 1, the *determineBestRoute* task will be represented by a *determineBestRoute* task-service, where the task is to determine the best route which can be provided by different services like eg. mapquest, map24, or a local authority of the emergency scenario. All services provide a route service although QoS attributes as well as the actual input and output may differ slightly. In particular, in the emergency case the local authority may provide the best information within their territory, while routes to get initially to the specific area of the emergency can be better provided by other service providers.

3.2 The Graphical Composition Language

The task-service construct is the basic building block of our composition language, which has been implemented in the SERCS tool. In the following we present the main concepts of the composition language depicted in the right part of Figure 1. We use graphical symbols to represent the metamodel concepts and a dotted line shows the relationship for three main concepts of the task-service construct. We present the provided constructs in three groups.

Regular nodes. This group contains two node types: task-graph and task-service (section 3.1). The task-graph node consists of a task part and an entire subgraph. Thus we have a construct that can be repeated at arbitrary many levels to create a recursive decomposition structure. A subgraph consists of two or more nodes, which are either task-service or task-graph nodes. All leaf nodes are task-service nodes.

Control flow. This group contains initialNode, finalNode, sequence, and-split, and-join, xor-split, xor-join. These well-known basic control flow constructs have quite logical names and a normal interpretation. An initialNode has exactly one outgoing edge, and a finalNode has exactly one incoming edge. There is exactly one initialNode and one finalNode for each graph. A finalNode within a subgraph terminates only the flow of the subgraph and the flow continues in the enclosing graph. The outermost finalNode terminates the whole composition execution. Furthermore, a regular node must have exactly one incoming edge and one outgoing

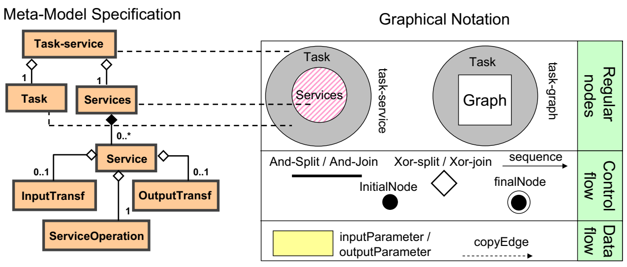


Fig. 1. The composition language

edge. This means that all the control flow is handled explicitly by the control flow constructs listed above. The proposed model represents a combination of structured flow models because each graph must have exactly one starting and one single ending node, and arbitrary models because no restrictions on the combination of control flow constructs are given [7].

Data flow. This group contains copyEdge, inputParameter, outputParameter and dataTransformation. Any number of inputParameters and outputParameters may be associated with the task parts of task-graph and task-service nodes, a service in a task-service node, and with dataTransformation services. A parameter has a name and a type which covers the syntactic definition and offers support for a semantic description. Parameters cannot appear as standalone objects and the set of inputParameters and the set of outputParameters (associated with an object) are unordered. A copyEdge connects a source parameter to a target parameter, and implies a deep copy. There may be an arbitrary number of outgoing copyEdges from a parameter, but there may only be one incoming copyEdge to a parameter. Furthermore these copyEdges may be connected between parameters at different nesting levels, which means that there may be a copyEdge from a parameter to another parameter within a subgraph. A data transformation has inputParameters and outputParameters. Each service in a task-service node is associated with two specialized data transformations: inputDataTransformation and outputDataTransformation. The dataTransformations act as mediators between the task and its services within a service-node. Data transformation techniques are a large matter on its own which we do not have enough space to explore in this paper. We assume that some kind of apparatus is available to do so. We also allow dataTransformations to be applied to inputParameters and outputParameters involving tasks only, but this is not important for the scope of this paper.

Now that we have introduced a composition language, based upon the task-service construct, we may take a closer look at the construct. We could require that realizing services for a task (in the task-service construct) were perfect matches so that the inputs and outputs corresponded one-to-one in both numbers, types and semantics. We feel that this would be too rigid, and that we would exclude a number of relevant services with only minor data format mismatches. This could be so

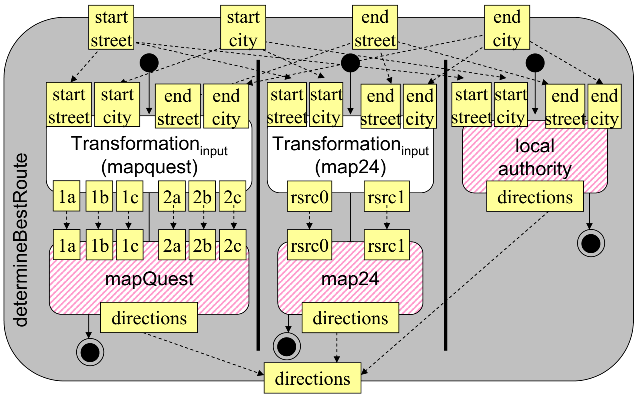


Fig. 2. A task-service example

simple that two different services have the same logical data input, but their syntactic XML tags are different. When we allow such mismatches to occur we need to use the transformation service on input and on output to build a bridge between a task and its service.

To illustrate the introduced concepts, we apply the definition on the scenario in Section 1. The task *determineBestRoute* (see Figure 2) should be operated on the input parameters of the starting street and city as well as the destination street and city. The output of the task is a route description. The *determineBestRoute* task can be implemented by services from mapquest, map24, or a local authority. Since these services are right now not directly provided as Web services we use the corresponding input and output parameters of the Web forms of the corresponding sites. All services have a single output parameter containing the route description. However, the input parameters required by the

– mapquest are 1a(2a) for the street, 1b(2b) for the zip code, and 1c(2c) for the city of the start and the destination respectively;
– map24 are rsrc0 and rsrc1 for the start and destination address respectively including all information;
– local authority are street and city of start and destination respectively.

As a consequence, a transformation service for mapquest and map24 is needed, while direct copyEdges can be used for the local authority. Figure 2 illustrates the scenario, where the different alternative services are depicted next to each other in the figure. Be aware that this is not a correct task-service construct since there are several initial and final nodes. However, we decided to depict it in such a way to represent the set of services implementing a particular task. In addition, the example covers several principles of our approach while others like e.g. graphs could not be considered due to the space limitations.

4 Using the Composition for Service Discovery

This section explains how the composition can be used to search for services. The main idea is that we look at the task parts only and ignore the service parts, because the task part contains all required information for the search. In our composition language there are only two kinds of regular nodes available, task-graph and task-service. Therefore, in the transformation to query documents we introduce a plain task node, which contains a single task node only. Plain task nodes are only used in transformed composition models and will never appear in an editable composition graph. We propose to perform two sequential steps which separate processing requirements from processing preference criteria:

1. *Search.* At first, a search is performed for each task-service node. The task part is transformed into a proper query denoting the requirements about the requested service covering syntactic interface definitions, semantic descriptions, and QoS constraints. How to actually represent such information in a query language will differ for each application case. Currently, there is no de-facto standard for such a query language. If the search does not identify suitable services, either the requirements must be reconsidered or the required functionality must be implemented.
2. *Select.* If more than one service is available that matches the requirements, a sorted list is desired where one can start with the closest matches first. In addition, QoS preference criteria can be considered selection criteria, e.g. to select the cheapest service. This step can include the negotiation and establishment of a QoS contract with the providers of the candidate services. This step will end with a chosen list of services.

After the appropriate services have been identified, the composition must be updated. All the necessary binding details of the chosen services must be registered within the service part of the corresponding task-service node for which we performed the search.

5 Using the Composition for Execution

This section shows how the outermost task-graph node (representing the entire composition) can be transformed into an executable document. In the transformation to an executable document we introduce the plain service node, which contains a single, executable service operation only. Plain service nodes are only used in the transformed composition models and will never appear in an editable composition graph. We assume that at least one service is identified for each of the task-service nodes. An algorithm comprised of two main steps are used to transform the graph into a one-level graph (no subgraphs) with service nodes only, and no tasks (except for the main composition itself which is a task-graph node).

Step 1: Replace task-service nodes by an explicit subgraph structure (transitions labeled 1 in Figure 3)**.** The task-service node defines an implicit structure which we will transform into an explicit structure (using our composition language) in this transformation step. We must assume that the necessary InputTransformation and the OutputTransformation specifications are defined already and are associated with the service. If not, then the specification is only partial and we cannot generate a fully executable document. (If the data transformations are omitted, we assume that the service is a perfect match of the task, and the control flow becomes trivial.) The task-service node is replaced by a task-graph node, where the task part remains the same and the subgraph is produced as described in the following text. If there is a single service inside the task, then the following sequentially ordered subgraph is produced: initialNode, InputTransformation, ServiceNode, OutputTransformation, finalNode. If there is more than one service (S1 in Figure 3), then we introduce an and-split after the initialNode and an xor-join before the finalNode. There will be parallel branches in between with the sequentially ordered triple InputTransformation, ServiceNode and OutputTransformation for each service. Notice that we allow an and-split to be followed by an xor-join, meaning that control-flow continues when the first parallel flow arrives and the other flows, produced in the and-split, are terminated or ignored. The decision on executing the alternative services in parallel has been made to achieve robustness of the implemented services. This approach is applicable as long there is no cost associated with the alternative services, otherwise this approach is far too expensive. A more detailed discussion of robustness and its implications can be found in [6].

Step 2: Flattening subgraphs (transitions labeled 2 in Figure 3)**.** This transformation step can be applied to all task-graph nodes except the outermost node representing the whole composition. We flatten a task-graph node by replacing it with the entire subgraph (except the initialNode and its outgoing edge, and the finalNode and its incoming edge) inserted into the parent graph. This is possible since we only allow graphs with a single outgoing edge from the initial-Node, and a single incoming edge to the finalNode. Notice that we also remove the task part of the original task-graph node in the process. The removal of the task part will be semantics-preserving with respect to the control and data flow. We do not show a proof of the claim in this paper, but illustrate in Figure 4 that the task part is redundant for the execution logics of task-service node. The input and output parameters of the task part are redundant since they are simply copied to and from the transformation services on input and output. Thus we can detach the task part and attach all its incoming and outgoing data flow (labeled `df-in` and `df-out`) and control flow edges (labeled `cf-in` and `cf-out`) to the transformation services instead.

We need to repeat steps one and two until there are no more task-graph nodes, and no more task-service nodes left. It is trivial to see that this approach can be supported by an algorithm that is guaranteed to terminate. The following observations should be sufficient to convince the reader; None of the two steps introduce task-service nodes; The first step reduces the number of task-service nodes; The second step reduces the number of task-graph nodes.

We have now produced a composition graph which is flattened, and that only uses basic control and data flow constructs, data parameters and a service node representing an executable service. These language concepts are supported by

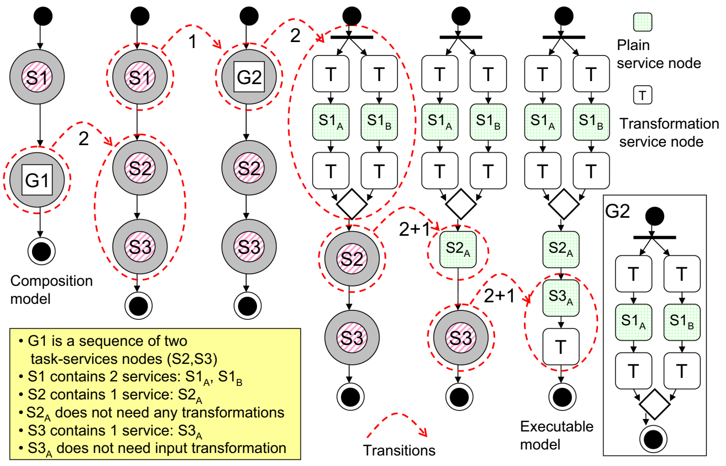

S1
G1
Composition model
S1
S2
S3
1
2
G2
S2
S3
2
T
S1A
S1B
2+1
Plain service node
Transformation service node
G2
Executable model
Transitions
• G1 is a sequence of two task-services nodes (S2,S3)
• S1 contains 2 services: S1A, S1B
• S2 contains 1 service: S2A
• S2A does not need any transformations
• S3 contains 1 service: S3A
• S3A does not need input transformation

Fig. 3. Example: Transformation to an executable model

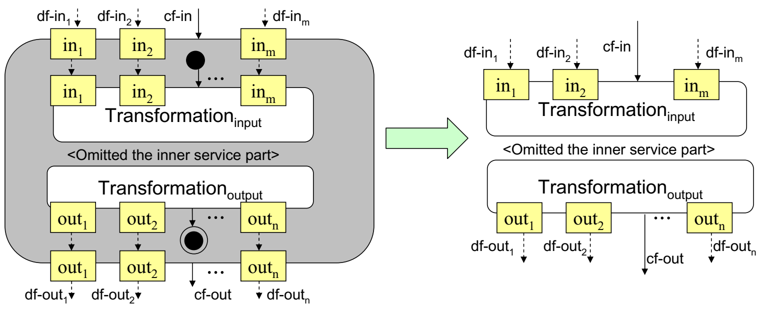


Fig. 4. Task parts are removed as part of the collapsing of subgraphs

most of the service composition languages and thus we believe that our composition language is transformable to most of these languages.

Figure 3 shows how a composition graph is transformed. The main task-graph node representing the whole composition surrounds the first subgraph structure and it is not shown in the figure. The first subgraph contains two regular nodes to be executed in sequence: a task-service node with the service part named S1, and a task-graph node with the subgraph named G1. Further details necessary to carry out the transformation are given in the text box at the bottom of the figure. Seven transitions (The last four transitions in the figure are combined into two) are needed to produce the final executable graph structure containing only plain services (including transformation services) and common control-flow constructs. For each transition we mark by an adjacent number which of the two algorithm steps are used. The rightmost box shows the subgraph structure named G2, which is produced as an intermediate step in step two of the example transformation.

6 Implementation in SERCS

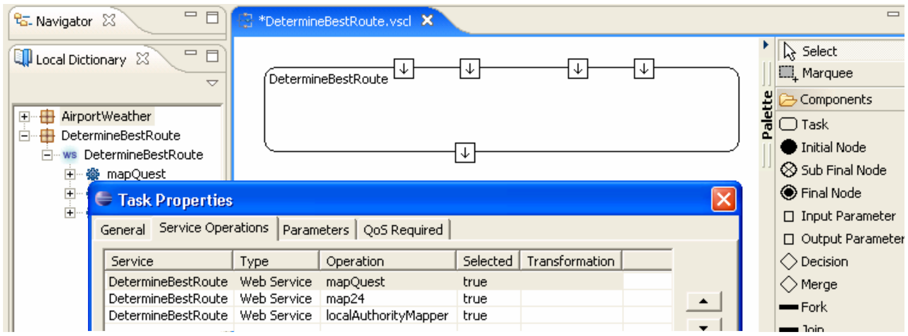
Our approach has been validated by implementing a tool called the Service Composition Studio (SERCS, pronounced *circus*) with the service composition language as its core. The design principle of the language is built around the task-service construct presented in this paper. Furthermore it is based on UML 2 activity models [9] regarding both graphical layout and most of the core concepts. The language can be viewed as a UML profile in that there are several extensions for QoS and semantic information as well as the enforcement of the task-service node construct. SERCS is an Eclipse-based tool with a GNU Public License. The graphical user interface of SERCS offers functionality to develop compositions, search for composition-relevant services and run service compositions. Figure 5 shows a model of *determineBestRoute* in the SERCS tool.

SERCS provides the ability to transform the task part of a task-service node in SERCS into USQL documents [16], perform a call to the externally defined USQL engine [16] to search for services, and to bring the found services back into the service part of the corresponding task-service node. The transformation is defined by XSLT with SERCS composition files as input and USQL documents as output. The transformation definition follows the approach described in Section 4 of searches for a single task in isolation, since this is the only alternative currently provided by the USQL engine.

SERCS provides the ability to apply an XSLT-defined transformation from a SERCS composition into a USCL document [10], and to perform a call to the externally defined JOpera engine [10] which executes the composition. USCL has its own sub composition construct which can be used to directly support the task-graph node. Thus we skipped step two of the algorithm in Section 5.

We have introduced a new graphical tool since existing languages do not support the task-service construct. We could have extended existing tools, but then we would not be able to enforce the use of the task-service construct. SERCS compositions are independent of the actual query and execution language. Thus alternative transformations could have been defined, such as a transformation from SERCS compositions to BPEL as the target execution language.

A running service composition, to be used in emergency situations, has been developed by the SERCS tool. A person at the emergency location uses a mobile phone to call the emergency help desk. Based on the mobile phone position, a number of steps is performed which finally delivers a map displaying the optimal driving route to the closest available ambulance. This composition consists of five task-graph nodes, fourteen task-service nodes, fifteen Web services, one Grid service, one Peer-to-Peer service and a number of data transformation services.



Service	Type	Operation	Selected	Transformation
DetermineBestRoute	Web Service	mapQuest	true	
DetermineBestRoute	Web Service	map24	true	
DetermineBestRoute	Web Service	localAuthorityMapper	true	

Fig. 5. Screenshot of the example within the SERCS tool

7 Discussion

Section 4 and Section 5 show that the requirements (final part of Section 2) of service discovery and service execution are satisfied by our composition language. The requirement of a single structure is satisfied since the task-service construct enables us to use a single graph for representing tasks and services.

A possible limitation with our approach is the manual decomposition. The composition developer manually proposes a decomposition which has a set of leaf tasks. It may be the case that a leaf task, T1, cannot be realized by a single service but that there are two services, S1 and S2, that can be executed in sequence to realize the leaf task. In our approach there is no way to determine this automatically. However, it may be that the search results for T1 returns S1 and/or S2 indicated to be partial matches, and that these give the composition developer a clue to revise T1 into a new subcomposition graph.

In our composition language we require that all services in a composition model belong to a task. This may seem too rigid and bothersome for cases where we know in advance exactly which service we want to use. The SERCS tool allows the user to insert such a service directly into the composition and the tool will automatically generate a task-service node with a task part based on the imported service. This will result in a task which initially lacks QoS requirements. It may also lack semantic descriptions if the imported service is not semantically described. We do allow for tasks that lack QoS and even semantic requirements, because currently the major part of service providers does not provice such description. It may even be argued that it is more relevant to associate QoS requirements only with the outermost task since the aggregated QoS is more interesting than how it distributes to the individual parts.

The automatic task production based on an imported service leads naturally to a question: *Are the explicit tasks redundant since they could be automatically generated by a transformation tool whenever we want to search for services?* Although this may work fine for some composition examples, we think that this in general is not satisfactory. This is because the automatic deducible task may be a poor task since it depends strongly on how well the service is described semantically. In many cases the semantic information may be missing or it may be that its interface is too specialized leading to missing matches of relevant services. In some cases it will also be relevant with local QoS requirements. Consider a service composition with several tasks of which one of them deals with a payment transaction. We may want to force a local QoS restriction of good security and high encryption level only to the payment service. Such local QoS requirements cannot be automatically deduced from a service since a service in the best case only advertises its own QoS offering.

The Web Service Modelling Ontology (WSMO) [8] contains description languages for tasks (Goals), services (Webservices) and a mapping (such as our DataTransformation) between task and service with the wgMediator. Up to our knowledge there is however no tool that enforces the coupling between a WSMO task and a service. Thus WSMO services may exist without a defined task, and there may be tasks which are not aware of existing mappings to services. This is a limitation for iterative service composition development.

The discussion of task-service right now focuses on stateless services, that is, a service associated to a task represents a service with a single request-response communication. Statefull services are maintaining an internal state and require several request-response communications, which may require an extension of the proposed approach on service discovery. We leave this to future work.

8 Conclusions and Future Work

Our contribution is a task-service construct which introduces a strong coupling between the definition of *what* we want to accomplish with *how* it can be accomplished by existing services. The task part defines the requirements for what we want to accomplish, and the service part defines all the discovered services that are capable of performing the task. When this task-service construct is the basic building block in a composition language, as within our SERCS tool, we achieve the benefits of maintaining a single graph which can be used both for service discovery and for executing service compositions. This benefit is crucial in an ever-changing SOA environment. We advocate for an iterative composition development with regular searches for newly introduced services. The service composition should take advantage of these new services and strive to always find the most appropriate services based on their QoS offerings.

Further research is needed to explore the effects of run-time search and selection on performance. Important topics to investigate for such run-time handling include trust, semantic precision and failure handling.

Acknowledgement. The work of Roy Grønmo has been partially supported by the European Commission under the contracts IST-FP6-004559 (SODIUM) and FP6-26514 (SWING).

References

1. Agarwal et al. A service creation environment based on end to end composition of Web services. In *International conference on World Wide Web*, 2005.
2. B. Ludäscher et al. Scientific Workflow Management and the Kepler System. *Concurrency and Computation: Practice & Experience, Special Issue on Scientific Workflows*, 2005.
3. BPMI.org. Business Process Modeling Notation (BPMN) Version 1.0, May 2004.
4. J. Cardoso and A. P. Sheth. Semantic E-Workflow Composition. *Journal of Intelligent Information Systems*, 21(3):191–225, 2003.
5. David L. Martin et al. Bringing Semantics to Web Services: The OWL-S Approach. In *Revised Selected Papers of the Intl Workshop Semantic Web Services and Web Process Composition (SWSWPC'04)*, San Diego, California, USA, July 2004.
6. M. C. Jaeger and H. Ladner. A Model for the Aggregation of QoS in WS Compositions Involving Redundant Services. *Journal of Digital Information Management*, 4(1):44–49, March 2006.
7. B. Kiepuszewski, A. H. M. ter Hofstede, and C. Bussler. On Structured Workflow Modelling. In *Proceedings of the 12th International Conference on Advanced Information Systems Engineering (CAiSE'00)*, volume 1789 of *LNCS*, pages 431–445, Stockholm, Sweden, June 2000. Springer Press.
8. R. Lara, D. Roman, A. Polleres, and D. Fensel. A Conceptual Comparison of WSMO and OWL-S. In *Web Services, European Conference (ECOWS 2004)*, September 2004. Erfurt, Germany.
9. O. M. G. (OMG). UML 2.0 Superstructure Specification, OMG Adopted Specification ptc/03-08-02, August 2003.
10. C. Pautasso and G. Alonso. The JOpera visual composition language. *Journal of Visual Languages and Computing (JVLC)*, 16(1-2):119–152, 2005.
11. J. Peer. A PDDL Based Tool for Automatic Web Service Composition. In *Proceedings of the Second Intl Workshop on Principles and Practice of Semantic Web Reasoning (PPSWR)*, St. Malo, France, September 2004.
12. S. R. Ponnekanti and A. Fox. SWORD: A Developer Toolkit for Web Service Composition. In *Proc. of the Eleventh International World Wide Web Conference (WWW*, Honolulu, Hawaii, USA, 2002.
13. Satish Tatte (Editor). Business Process Execution Language for Web Services Version 1.1, February 2005.
14. F. Schiersner. Fallstudien: Die Oder-Flut im Sommer 1997. http://www.krisennavigator.de/kafa1-d.htm.
15. P. Traverso and M. Pistore. Automated Composition of Semantic Web Services into Executable Processes. In *The Semantic Web - ISWC 2004: Third International Semantic Web Conference,Hiroshima*, Hiroshima, Japan, November 2004.
16. A. Tsalgatidou, M. Pantazoglou, and G. Athanasopoulos. Specification of the Unified Service Query Language (USQL), Technical Report, June 2006.
17. W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.