Framework Managing the Automated Construction and Runtime Adaptation of Service Mashups

Anna Hristoskova, Bruno Volckaert and Filip De Turck

Department of Information Technology, Ghent University - IBBT
Gaston Crommenlaan 8 bus 201, B-9050 Ghent, Belgium

Abstract. With an increased deployment of new software services, reusing existing ones as building blocks to create new service mashups offers flexibility to the developer and accelerates the design process. In this way businesses are able to create value at reduced development time and cost.
In order to allow for the automation of this emerging engineering methodology the paper presents a framework for the construction of new applications without the intervention of the ICT department. This framework offers the needed support through the use of planning algorithms automatically combining semantically enriched services into new mashups. The developed algorithms are optimized with runtime adaptation to changing user-context taking fully into account the provided quality of service parameters of the available building blocks.
The enrichment of the available business services with semantics, reasoning, and at-runtime composition are evaluated by means of a framework providing a management interface for an e-shop application.

1 Introduction

Instead of building self-contained silos, businesses break down their applications in independent components offering a scoped functionality using open coding and communication standards. The creation of catalogues of reusable components means agile construction of new applications and faster adaptation to the changing business environment. These service mashups combine functionality and content from existing sources, creating greater value than the sum of the individual participating components. Currently, Web services are the most adopted technologies for constructing mashups. Designed to support interoperable machine-to-machine interaction over a network, they are capable of being accessed via standard protocols such as SOAP over HTTP.

Service-Oriented Architectures [1] offer the advantage of building component-based systems using Web services. In a dynamic environment where the desired functionality cannot always be predicted, all kinds of custom made compositions can be built from scratch meeting the users' requests without the need for continuous interaction between users and developers. A computer-aided automatic approach is possible through the adoption of ontologies and the Semantic Web [2] for the enrichment of services

Hristoskova A., Volckaert B. and De Turck F..
Framework Managing the Automated Construction and Runtime Adaptation of Service Mashups.
DOI: 10.5220/0003350300310043
In *Proceedings of the International Workshop on Semantic Interoperability* (IWSI-2011), pages 31-43
ISBN: 978-989-8425-43-0

with machine-processable semantics. Specifications such as OWL-S [3] provide a semantic description for Web services defining **i**nputs, **o**utputs, **p**reconditions and **e**ffects (IOPEs), and nonfunctional properties. Thanks to these descriptions a computer system would be able to automatically combine services together executing a more sophisticated task provided for the implementation of reasoning and matching algorithms.

The proposed framework in this paper offers an environment for the automatic construction and execution of service mashups departing from available functionality found on the Web and within enterprises. It disposes of a user interface for the management of semantically annotated services and the definition of users' requests (e.g. application manager, end-user). Planning algorithms are designed for the construction of service mashups solving these requests using the available resources. Novelty is the framework enhancement with at-runtime adaptability anticipating changes (e.g. availability of new services, resource and service failures) and resolving decision points in the composition through the use of control constructs and user-defined business logic rules.

The remainder of the paper presents in Section 2 the architecture of the mashup creation and execution framework whose components are detailed in Section 3. Following is an evaluation of an e-shop management application in Section 4. Section 5 gives a discussion of the current research in this field. Finally, the main conclusions are stressed in Section 6 and new possibilities for enhancing the framework are explored.

2 Framework Architecture and Process Flow

Figure 1 presents the mashup creation and execution architecture detailed in [4]. The **Configuration Frontend** provides the application manager with a user interface for the definition of requests (goals and business logic rules) and the management of the available services and their quality attributes. All requests are handled by the **Coordinator** which is based on a Microkernel pattern. It manages the service repository, the generation of goals from users' requests, the necessary configurations before processing of the requests, and the communication of the mashup creation and execution state back to the Frontend.

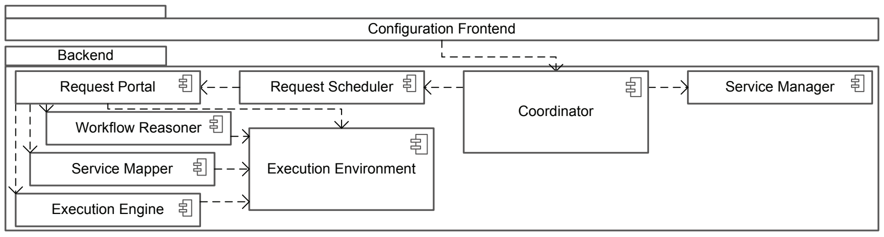


Fig. 1. Main building blocks of the mashup creation and execution framework.

The mashup composition and execution process is presented in Figure 2. Starting from a semantic definition of a user-defined goal, a *service mashup* is constructed from the available services by the **Workflow Reasoner**. The inner planning algorithms and semantic matching techniques of this module will be further detailed in Section 3. Next the **Service Mapper** converts the constructed mashup into an executable process by

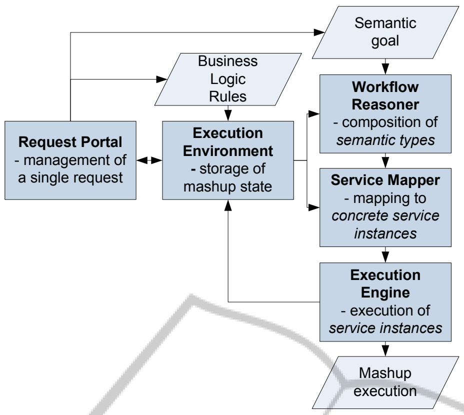
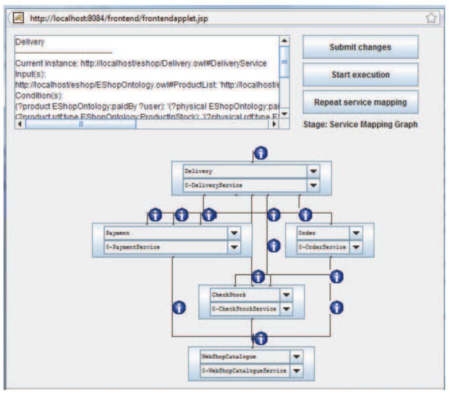

Semantic goal
Business Logic Rules
Workflow Reasoner - composition of semantic types
Request Portal - management of a single request
Execution Environment - storage of mashup state
Service Mapper - mapping to concrete service instances
Execution Engine - execution of service instances
Mashup execution

Fig. 2. Mashup creation, execution and runtime adaptation process for a user's request.

selecting the specific *service instances* for each building block of the mashup [5] satisfying the predefined quality of service constraints and requirements (execution time, cost) and defining the necessary bindings between them. The process is executed by the **Execution Engine** handling the invocation workflow of the service instances by forwarding the results to the right components. The **Execution Environment** acts as a storage for users' requests, business logic rules (desired service instances), inputs, results, execution state of the service mashup. The Reasoner and Mapper utilize this data (e.g. intermediary results) to optimize the reasoning and mapping process of the service mashup at design and runtime. The reasoning process is divided into two steps. First through backward chaining a generic composition of services is constructed specifically resolving user-defined goals. A forward chaining procedure further tunes this composition utilizing the stored data in the Environment.


http://localhost:8084/frontend/frontendapplet.jsp

```
Delivery
-----------------------------------
Current instance: http://localhost/eshop/Delivery.owl#DeliveryService
Input(s):
http://localhost/eshop/EShopOntology.owl#ProductList: 'http://localhost/e
Condition(s):
(?product EShopOntology:paidBy ?user): '(?physical EShopOntology:pai
(?product rdf:type EShopOntology:ProductInStock): '(?physical rdf:type E
```

Stage: Service Mapping Graph

Fig. 3. Configuration Frontend, showing a composite mashup graph.

A **Request Portal** provides a management interface keeping track of the whole process for a single user's request. It visualizes the service mashup, the utilized resources for execution, and returns intermediary results via the Frontend. The user interacts with the system through the interface in Figure 3 where he can tune the constructed mashup and change the selected resources to his specific preferences.

3 Details of the Reasoning and Composition Process

This section focuses on the novel contributions of the framework starting with a discussion on the semantic grouping of equivalent service instances. Following is an overview of the matching possibilities between these building blocks during the composition process. Furthermore, the actual composition process is detailed together with the at-runtime reconfiguration and adaptation methods.

3.1 Semantic Description of Service Instances

A distinction is made between two types of building blocks used by the presented framework: *concrete service instances* and *abstract semantic types.*

Concrete services instances are the actual services executed on specific resources. Each *service instance* is provided with several QoS (Quality of Service) parameters describing its properties. Examples include the average execution time, economic cost, and availability. These QoS parameters are defined beforehand or their values are dynamically adjusted based on previous invocations. Thus, for example, the average execution time is updated after the invocation of the service. The QoS of a specific service instance consists of a QoS Type, QoS Value, and QoS Comparator. A QoS Type can amongst others be the economic cost for executing a service, the execution time. Each QoS Type has a QoS Value and a specific QoS Comparator for comparing the actual QoS Values. This offers an application manager with the possibility to extend the framework with new QoS parameters and define customary comparison techniques.

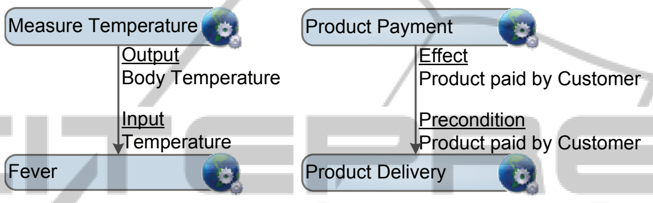
In the presented framework, these *service instances* are enriched with semantic annotations using OWL-S. As several semantically equivalent *service instances* (equivalent IOPEs) exist, their semantic interfaces are grouped into a single *semantic type*, thus reducing the search space of available instances. For example, multiple payment services (bank transfer, Visa, PayPal) are grouped into one semantic payment type.

The OWLS-MX Matchmaker [6] provides a partial solution to this problem by comparing service inputs and outputs and assigning a score based on the semantic distance between these concepts. As a truly equivalent match between these service interfaces cannot be expected, the services are grouped in a hierarchical fashion. Although exhaustive enough, the OWLS-MX solution lacks the ability to compare service preconditions and effects. Therefore, we extended this approach in order that during the composition of the service mashup the Workflow Reasoner is able to search for a specific group of services producing required outputs and more importantly effects (detailed in Section 3.3). Afterwards the Service Mapper will select a corresponding *service instance* offering required QoS.

3.2 Semantic Match between Semantic Types

Semantic types are compared and linked by the Workflow Reasoner in case of matching input-output and/or precondition-effect relations. Depending on the quality of the match between their interfaces, control constructs are required for the construction and more importantly execution of more complex service mashups. This section gives special attention to the use of 'IfThenElse' and 'ForEach' control constructs of OWL-S.

Parametric Match. As OWL-S is used for describing Web services, the service inputs and outputs are expressed by OWL concepts. We define an input-output match between services when an input and an output represent similar semantic concepts and as a result the output of one of the services is used as input for the other. As demonstrated in Figure 4(a), the output 'Body Temperature' is interpreted as a kind of 'Temperature', matching the measuring service to the service determining the patient's fever.



(a) Input-Output match (b) Precondition-Effect match

Fig. 4. Semantic match between two service interfaces.

We define different qualities of semantic matches between service inputs and outputs, some of which require additional control constructs:

– **Exact.** The service output exactly matches the semantic concept of the service input.
`S:WebShopCatalogue(O:Product) -> S:Delivery(I:Product)`
– **Subsume.** The output concept inherits from the input. This is a valid but lower quality match.
`S:WebShopCatalogue(O:PhysicalProduct) -> S:Delivery(I:Product)`
– **Relaxed.** The service input is more specific than the output concept in which case this is not a valid match. It is nevertheless incorporated in the composition process as a generic output can turn out to be a more specific individual after service execution. For instance, a shopping basket may consist of digital and/or physical products in which case some products need to be downloaded and others delivered. This is resolved through the use of a repository describing product individuals and their inheritance graph. In this case, the Workflow Reasoner adds an 'IfThenElse' construct between the matching services with an 'If'-condition on the specific product type output.
`S:WebShopCatalogue(O:Product) -> S:Delivery(I:PhysicalProduct)`
– **List.** The service output is a list of concepts matching the single input concept. For example, a shopping basket used as input for a service checking the stock status of each individual product. Here, a 'ForEach' construct is added iterating over the list of products. The specific match for each concept from the list can be all of the above (**exact**, **subsume**, **relaxed**).
`S:WebShopCatalogue(O:ProductList) -> S:Delivery(I:PhysicalProduct)`

Condition Match. For the definition of the service preconditions and effects we use SWRL (Semantic Web Rule Language) [7] expressions and built-ins (SWRLB) such as comparisons. An SWRL expression consists of a property and one or more arguments (semantic OWL concepts). Examples include an `OWLClass` with one argument[1], an `ObjectProperty` with two arguments[2], a `DataProperty` with one argument and an RDF type[3], a SWRLB primitive with one or more arguments[4]. We define a precondition-effect match when the result of the execution of a service corresponds to a condition required for the execution of another service. The effect in Figure 4(b) of the product payment service realizes the payment condition for the delivery of the product to the customer.

Similar to the input-output match, different qualities of semantic matches between service preconditions and effects are defined together with the necessary control constructs:

- **Exact.** The service effect exactly matches the SWRL expression representing the service precondition. A matching SWRL expression consists of an *equivalent* property and *exactly* matching semantic concepts (arguments of the property).
  `S:CheckStock(E:Product InStock) -> S:Delivery(P:Product InStock)`
- **Subsume.** The effect property still matches the precondition property and between the semantic concepts a *subsume* match is defined. For an ObjectProperty only the first argument can be a subtype, however if the property is defined as an inverse property, the second is also a subtype as OWL concepts inherit the properties of the super class.
  `S:CheckStock(E:PhysicalProduct InStock) -> S:Delivery(P:Product InStock)`
- **Relaxed.** This is similar to the *subsume* match, however one still needs to check if the concepts can be subclassed after execution as is the case in the *relaxed* input-output match.
  `S:CheckStock(E:Product InStock) -> S:Delivery(P:PhysicalProduct InStock)`
- **Conditional.** It should be noted that a service effect can be conditional; meaning that depending on the service output a different effect is possible. For example a service checking the stock of a product can have as effect that the product is in stock *if* the stock status is true or not in stock *if* false in which case an ordering service is added to the composition before the actual delivery of the product. In this case an 'IfThenElse' construct is added with the conditional output on the effect as 'If' statement. The following is an example of the conditional effect of the CheckStock service where the product is considered in stock if the stock status is true:

```
<process:hasResult>
   <process:inCondition>
      <expr:SWRL-Condition rdf:ID="ProductInStockCondition">
        swrlb:equal(eshop:stockStatus,rdf:boolean(true))
      </expr:SWRL-Condition>
   </process:inCondition>
   <process:hasEffect>
      <expr:SWRL-Expression rdf:ID="ProductInStockEffect">
```

[1] InStock(Product)
[2] paidFor(Customer, Product)
[3] hasName(Customer, String)
[4] equal(StockStatus,true)

```
            eshop:ProductInStock(eshop:PhysicalProduct)
          </expr:SWRL-Expression>
        </process:hasEffect>
      </process:hasResult>
```

The matching strategies are used by the planning algorithms of the Workflow Reasoner during the construction of the service mashup where services are selected providing outputs and/or effects matching the required (service) inputs and/or preconditions.

3.3 Workflow Reasoner based on HTN Planning

A Hierarchical Task Network (HTN) plan [8], [9] is a partially ordered graph of service nodes. Each service defines a certain state (i.e. the outputs and effects of the service execution) and the description of the overall state is distributed in the graph. Services of unordered nodes (in parallel paths) are executed simultaneously, through the use of the 'Split+Join' construct of OWL-S.

The Workflow Reasoner in this paper adopts the HTN planning methods by incorporating the semantic grouping and matching strategies (including the use of control constructs) mentioned above. It is enhanced with runtime mashup adaptation using collected data in the Execution Environment (Section 3.4).

Planning proceeds as follows: the user's request goes through an *expansion* phase followed by the actual *construction* through semantic matching of services. The user-defined requests, consisting of initial and goal state (`RQ=IS+GS`), are transformed into provided inputs and valid preconditions (`IS=I+P`) and required outputs and effects (`GS=O+E`) resulting in an abstract semantic service description (i.e. IOPEs). During the *expansion phase*, this *abstract service* is split up in the outputs and effects (`O+E`) that need to be resolved and the inputs and preconditions (`I+P`) that can be utilized for this purpose. On one hand, an available *semantic service (mashup)* can be matched to the required interface immediately ending the composition stage. On the other hand, if no complete solution already exists, the *construction phase* generates a plan of services using backward chaining strategy transforming the goal state into the initial state (`GS->IS`). The selected matching services are queued and resolved in a breadth-first fashion. The inputs and preconditions of the service on top of the queue are linked to matching service outputs and effects (`O+E->I+P`) as described in Section 3.2. If necessary control constructs are added depending on the quality of the match. It is important to note that in case no matching services are found, an exhaustive composition is presented and the incomplete inputs/preconditions are marked. These are provided by the application manager and/or new services are deployed filling in the missing gaps.

3.4 Runtime Adaptation

An important aspect of the presented framework is the runtime behavior anticipating changes (e.g. availability of new services, resource and service failures) and adapting each request to user-specific business logic rules. Figure 5 presents a feedback principle where services are executed using inputs and conditions from the Execution Environment and new service effects and outputs are produced and added to this Environment. This results in a dynamic system where new knowledge is inferred at runtime.

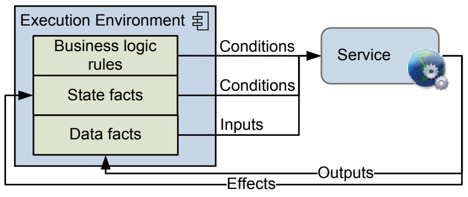


Fig. 5. Inference through matching of service preconditions and inputs and returning service effects and outputs.

Business Logic Rules. These rules, added offline or at runtime to the Execution Environment, are used by the Workflow Reasoner and Service Mapper to adapt the generic mashup from Section 3.3 to the users' needs at design and runtime. For example, there are several possibilities for product delivery services which we define as a decision point in the execution: a simple 'Delivery' service, 'DeliveryWithPaymentOnDelivery' service, 'DeliveryToProxy' service. At this point, the user has to select one of the possibilities. A business logic rule could be stating that the default service is the 'DeliveryWithPaymentOnDelivery'.

Iterative Pruning. Several mashup creation and execution iterations can be performed before the final execution, in case the constructed mashup graph is too complex (e.g. having too many branches, conditional paths, decision points). Our framework executes parts of the mashup, while trying to preserve the current state, by evaluating only the stateless services. The Execution Engine stores any intermediary results in the Execution Environment which are used to prune the mashup graph by resolving the decision points or conditional paths. The Workflow Reasoner collects the needed data in order to reconfigure the service mashup and remove parts of the branching. For example, if the stock of the selected product is checked beforehand, one can decide whether a product ordering service is needed or it can be removed from the constructed mashup.

Failure Recovery. The current state is stored in a similar fashion in the Execution Environment during the mashup execution. If a service fails, it is used by the Workflow Reasoner and Service Mapper to track the failed services and the current state of the system. During recovery the Service Mapper will select an alternative service instance equivalent to the failed one (same semantic group) and proceed with execution. If no alternative instance exists, the Workflow Reasoner will construct an alternative service composition replacing the failed point keeping in mind the current state of the executed mashup. In case that also fails the user will be notified of the specific component problem.

4 E-Shop Workflow Design

The presented framework is evaluated by means of a management interface for the automatic construction and runtime adaptation of e-shop applications. An e-shop ontology is created defining the concepts used for the annotation of the e-shop services in OWL-S.

4.1 Design of an E-Shop Application

A sale consists of a customer buying one or more products. This means that:

Trigger. A potential customer browses to the product catalogue of the e-shop.
Initial State. The e-shop and customer info is known. This includes account information necessary to make payments to the e-shop.
Goal Description. The composition is successfully executed, when the following effects are reached:

1. The customer ordered the product(s).
2. The customer paid for the product(s).
3. The product(s) was(were) delivered to the customer.
   - Digital products, such as music and software, are downloaded.
   - Physical products are transported to the customer's delivery address or to a proxy point of the customer's choice.

4.2 Construction of the E-Shop Workflow

For each required input and condition of an e-shop service, the Workflow Reasoner matches a corresponding service output and effect constructing an e-shop workflow. It keeps track of the conditional paths so that during the construction of the executable process the Service Mapper will add, if required, control constructs. Figure 6 presents the workflow of the different e-shop services from selection to payment and delivery of the products. The effect of the selection is implied by the output of the 'WebShopCatalogue', which represents a list of selected products. If the customer fails to select one or more products, the execution of the composition is prematurely ended, otherwise a 'ForEach' construct iterates over each product. A decision ('IfThenElse' construct) is made whether the product is in stock or should be ordered followed by 'Payment' and 'Delivery'. A *Delivery method* is added having as result one or more payment and delivery options ('Choice' construct) through which, according to the configurable rules, the purchase is made. This result is not known at composition time but can be defined through business logic rules by the user, being a customer or an e-shop manager. If the result is a specific delivery method defined by the e-shop manager, the purchase is made in that way. If it is more than one, the customer chooses amongst all the possibilities and the execution path depends on his decision. The result of this interactive choice is not always known at composition time: the customer makes a choice after being presented with the different execution paths. Consequently, the e-shop workflow exposes a decision point where the correct branch is chosen at runtime and followed during execution.

Service Grouping and Composition Performance. For testing purposes, the e-shop mashup in Figure 6 was designed, consisting of 6 levels, breadth of maximum 3 services, and 10 different available service nodes multiplied by 5 semantically equivalent services per node. The composition time of the Workflow Reasoner was evaluated for growing number of equivalent services with or without service grouping. The results are presented in Table 1 including the time needed to load (and group) the services in

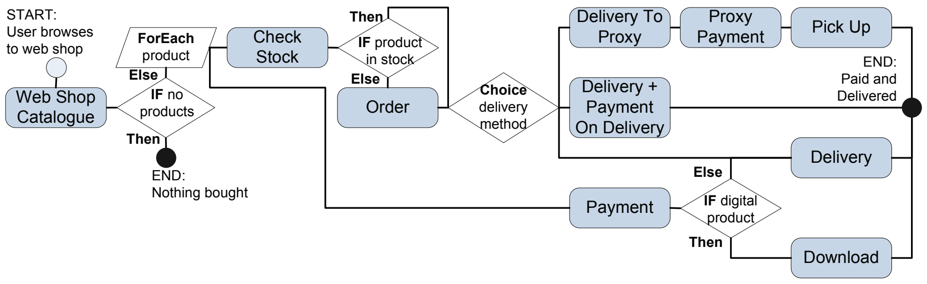


Fig. 6. Constructed e-shop workflow with the use of control constructs.

the repository. During the loading of the first service description, several other ontologies need to be loaded like the OWL-S Profile, Process, and Grounding ontologies, the specific use case E-Shop ontology, the SWRL rules ontologies. Once this is done, the only lost time is during the semantic matching of the service interfaces in order to group the equivalent services. Therefore while the difference between service loading with or without grouping grows up to a second, the composition time without grouping grows exponentially as all available services are considered for the workflow construction. With grouping only the groups of equivalent services are considered.

Table 1. Comparison of the e-shop composition time with or without service grouping.

	Service loading (ms)				Composition time (ms)			
	Without Grouping		With Grouping		Without Grouping		With Grouping	
# Services	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ	$\overline{x}$	σ
1	739	15	818	36	9064	49	8921	283
2	1122	89	1470	80	25238	514	9152	185
3	1452	49	2253	61	49563	624	9341	128
4	1875	118	3077	74	94362	1432	9656	249
5	2127	57	3928	31	151691	3271	10269	105

4.3 Runtime Adaptation of the E-Shop Workflow.

This section details the runtime behaviour of the framework as described in Section 3.4 for the e-shop workflow.

Business Logic Rules. In order to execute the e-shop workflow, the e-shop manager needs to define business logic rules expressing which 'Payment' and 'Delivery' method should be chosen or the customer should choose from the offered possibilities. On one hand the design time configuration by the e-shop manager defines the workflow permanently. On the other hand the choice made by the customer during invocation requires at-runtime adaptation. Once this choice is made, the reasoning process automatically configures the workflow through the removal of the decision point and the selection of only one 'Payment' and 'Delivery' path. For example if one defines 'Payment followed by Delivery' all the other options such as 'Payment on Delivery' and 'Delivery to Proxy' are discarded from the workflow.

Iterative Pruning. The e-shop workflow is further pruned through the execution of the effectless services. Depending on their output, further decisions are made, reducing the execution paths. For example, by executing the 'WebShopCatalogue' service, the Workflow Reasoner decides whether there are any selected products and if they are digital or physical. Then, the 'CheckStock' service verifies whether the physical products if any are in stock. This way the 'Download' or 'Delivery' and/or 'Order' services are automatically removed.

Failure Recovery. During the e-shop execution state information is recorded in case of a resource or service failure. For instance simultaneously to the execution of the product 'Payment', the 'Order' service fails. The Service Mapper will select an *equivalent* ordering *service instance* replacing the failed one. Afterwards the Execution Engine will avoid a repeated 'Payment' execution. On the other hand, if no equivalent ordering service instance is found, the Workflow Reasoner will reconfigure the original mashup constructing an alternative solution for the product ordering, treating the 'Payment' requirements as already met and thus as part of the initial state.

5 Related Work

Today a number of popular workflow standards and implementations [10], such as BPMN, BPEL4WS, XLANG, WSFL, still exhibit several shortcomings: no automatic or dynamic deployment support, limited reliability guarantees.

In [11] a predefined OWL-S workflow is first translated in SHOP2 syntax and then HTN planning is executed. SHOP2 does not support an output concept and OWL-S's 'Split' and 'Split+Join' control constructs so the system does not handle concurrency. OWLS-Xplan [12] constructs a service sequence, as opposed to a mashup graph, using an ontological definition of the initial and the requested goal state. However, before planning, the OWL-S 1.1 service descriptions are first converted to corresponding PDDL 2.1 (Planning Domain Definition Language) descriptions which could raise performance issues. The PDDL planner is in turn a linear STRIPS planner extended with HTN planning.

Several research projects some of which within the European Union Sixth and Seventh Framework Programme aim at creating platforms supporting the creation, management and execution of service mashups. Reservoir [13] combines virtualization and grid computing creating distributed service-oriented infrastructures. Platforms like INFRAWEBS [14] and Amigo [15] propose approaches, in which the process of finding appropriate services is guided by algorithms for decomposition of user goals into sub-goals and discovering the existing services able to satisfy these sub-goals without further planning. MashWeb [16] and SOA4All [17] focus on the creation of data flows controlling the output-input flows and workflows controlling the execution sequence of the services.

The presented framework in this article constructs service mashups starting from initial and goal state through matching of service effects to required preconditions. Planning is immediately performed in OWL-S, adopting the richness of the OWL-S control constructs such as 'Split+Join', 'IfThenElse', 'ForEach', 'Choice'. The framework is designed in a way that different Workflow Reasoners, QoS-aware Service Mappers and Execution Engines are easily plugged in just by extending the respective interfaces. Late binding is used to select the services offering the desired QoS for execution. Several (partial) iterations of mashup configuration and execution are possible as intermediary results are used as feedback to further tune the service mashup at design and runtime. The use of business logic rules defined by the user enables further tuning and personalization of his requests.

6 Conclusions and Future Work

This paper focuses on the design of a framework for the automated management of new applications through dynamic composition and execution of the building blocks of service mashups. Based on semantic descriptions of Web services, reasoning algorithms are developed for automatically composing new service mashups realizing defined goals. These algorithms define a planning system using control constructs based on the quality of the match between the semantic services. QoS constraints and requirements are satisfied through late binding to specific service instances. The system responds dynamically at runtime to changing context such as new business logic, new services, failure or overload of network elements or services. An e-shop case is implemented evaluating the proposed framework and illustrating the workflow execution optimizations.

In the future the planning and execution framework will be extended with a distributed deployment component which will execute the different service instances making optimal use of the available resources. Furthermore, techniques will be studied to take into account trends in user and resource behavior, in order to optimally design context-aware service mashups.

Acknowledgements

This work is partly funded by WTEPlus, an IBBT GBO project on the definition of an open architecture that allows the creation, sharing and composition of service mashups, seamlessly combining functionality found on the Web, the enterprise or within the ’walled garden’ of the telecom operator.

References

1. Papazoglou, M. P., Traverso, P., Dustdar, S., Leymann, F.: Service-Oriented Computing: State of the Art and Research Challenges, IEEE Computer Society, vol. 40(11), pp. 38–45, (2007).
2. Berners-Lee, L. , Hendler, J. , Lassila, O. : The Semantic Web: A New Form Of Web Content That Is Meaningful To Computers Will Unleash A Revolution Of New Possibilities, Journal of the Scientific American, vol. 284(5), pp. 34–43, (2001).
3. OWL-S, http://www.w3.org/Submission/OWL-S/ [Online].

4. Hristoskova, A., Volckaert, B., De Turck, F., Dhoedt, B.: Design of a Framework for Automated Service Mashup Creation and Execution Based on Semantic Reasoning, 2010 The Fifth International Conference on Internet and Web Applications and Services (ICIW 2010), pp. 149–154.
5. Avellino, G., Boniface, M., Cantalupo, B., Ferris, J., Matskanis, N., Mitchell, B., Surridge, M.: A Dynamic Orchestration Model for Future Internet Applications, ServiceWave 2008, LNCS, vol. 5377, pp. 266–274, Springer, Heidelberg (2008).
6. Klusch, M., Fries, B., Sycara, K.: Automated Semantic Web Service Discovery with OWLS-MX, In Proceedings of 5th International Conference on Autonomous Agents and Multi-Agent Systems, AAMAS 2006.
7. SWRL, http://www.w3.org/Submission/SWRL/ [Online].
8. Nau, D., Au, T., Ilghami, O., Kuter, U.,. Murdock, J, Wu, D., Yaman, F.: SHOP2: An HTN planning system, Journal of artificial intelligence research, vol. 20(1), pp. 379–404, (2003).
9. Hristoskova, A., Volckaert, B., De Turck, F.: Dynamic Composition of Semantically Annotated Web Services through QoS-Aware HTN Planning Algorithms, Proceedings of the Fourth International Conference on Internet and Web Applications and Services (ICIW 2009), pp. 377–382.
10. Van der Aalst, W. M. P., Dumas, M., ter Hofstede, A. H. M.: Web service composition languages: Old wine in new bottles, Proceeding of the 29th EUROMICRO Conference: New Waves in System Architecture, pp. 298–305, (2003).
11. Sirin, E., Parsia, B., Wu, D., Hendler, J., Nau, D.: HTN planning for web service composition using SHOP2, Journal of Web Semantics, vol. 1(4), pp. 377–396, (2004).
12. Klusch, M., Gerber, A., Schmidt, M.: Semantic Web Service Composition Planning with OWLS-Xplan, Proceedings of the First International AAAI Fall Symposium on Agents and the Semantic Web, (2005).
13. Rochwerger, B., Breitgand, D., Levy, E., Galis, A., Nagin, K., Llorente, I., Montero, R., Wolfsthal, Y., Elmroth, E., Caceres, J., others: The RESERVOIR Model And Architecture for Open Federated Cloud Computing, IBM Systems Journal, vol. 53(4), (2009).
14. Agre, G., Marinova, Z.: An INFRAWEBS Approach to Dynamic Composition of Semantic Web Services, Cybernetics and Information Technologies, vol. 7(1), pp. 45–61, (2007).
15. Valle, M., Ramparany, F., Vercouter, L.: Dynamic service composition in ambient intelligence environments: a multi-agent approach, Proceeding of the First European Young Researcher Workshop on Service-Oriented Computing, (2005).
16. Pfeffer, H.: A Underlay System for Enhancing Dynamicity within Web Mashups, International Journal On Advances in Software, vol. 2(1), pp. 63-75, (2009).
17. Lecue, F., Delteil, A., Leger, A.: Towards a Semantic State Transition System for Automated Generation of Data Flow in Web Service Composition, In International Journal of Semantic Computing (IJSC), vol. 3(4), pp. 499–526, (2009).