

Pløengine: A System for Automated Service Composition and Process Enactment*

Harald Meyer, Hagen Overdick, and Mathias Weske
Hasso-Plattner-Institute for IT-Systems-Engineering at the University of Potsdam
Prof.-Dr.-Helmert-Strasse 2-3, 14482 Potsdam, Germany
{harald.meyer,hagen.overdick,mathias.weske}@hpi.uni-potsdam.de

Abstract

An important challenge in software technology is the construction of infrastructures for the integration of functionality provided by multiple organizations. A promising approach is the service-oriented architecture that allows the offering of services in a uniform way and the composition of new services out of existing ones. Traditionally, this was a manual task. Recently, the automation of service composition has developed into an important research topic, as it promises to improve the efficiency of service composition as well as the flexibility of their enactment. In this paper we present design and implementation concepts of Pløengine, a software system that aims at supporting the planning of service compositions as well as their flexible enactment.

Categories: software architectures, processes and service composition, automated planning.

1. Introduction

An important challenge in software technology is the construction of infrastructures for the integration of functionality provided by multiple organizations [1, 2]. Service-orientation is a promising approach to achieve this goal, since functionality is provided through composable services. These services are specified in a uniform format, and they can be orchestrated using workflow technology [12, 9]. While composing services has traditionally been a manual task that requires considerable resources, recently automated service composition

using planning algorithms [5, 8] became an important research topic to improve the efficiency of service composition as well as the flexibility of their enactment [19, 20]. In this paper we present the Pløengine system that integrates the automated composition and flexible enactment of service compositions. This is achieved through an integrated platform architecture as well as a shared meta-model.

In recent years, Web Services technology in general and service composition especially has been mainly driven by industry, where the definition of a respective XML language (WS-BPEL [15]) has gained considerable interest. In WS-BPEL, the set of available services is stable and service requests are uniform, so that static environments as well as static process structures address production workflows [12]; tools to enact WS-BPEL processes are available. In these settings, service compositions can be modeled manually to comply to service requests. While this approach is valid in production workflow environments, it is rather problematic in highly dynamic systems where service requests and available services might vary over time. This is the case in service-oriented environments. As a result, manually modeled composed services can hardly incorporate a wide range of service requests and changes in the available services, so that an approach based on planning of service compositions is more promising.

The remainder of this paper is organized as follows: The challenges integrating the meta-model are elaborated in the next section. The architecture for an integrated platform for service composition planning and enactment and its prototypical implementation in the Pløengine system (<http://plaengine.com>) are presented in Section 3. This paper closes with an overview over related work in Section 4 and an outlook on future work and a conclusion in Section 5.

* This research is supported by the Adaptive Services Grid (ASG) project funded by the Sixth Framework Programme of the European Commission and the Process Family Engineering in Service-Oriented Applications (PESOA) project funded by the German Ministry of Research and Education (BMBF).

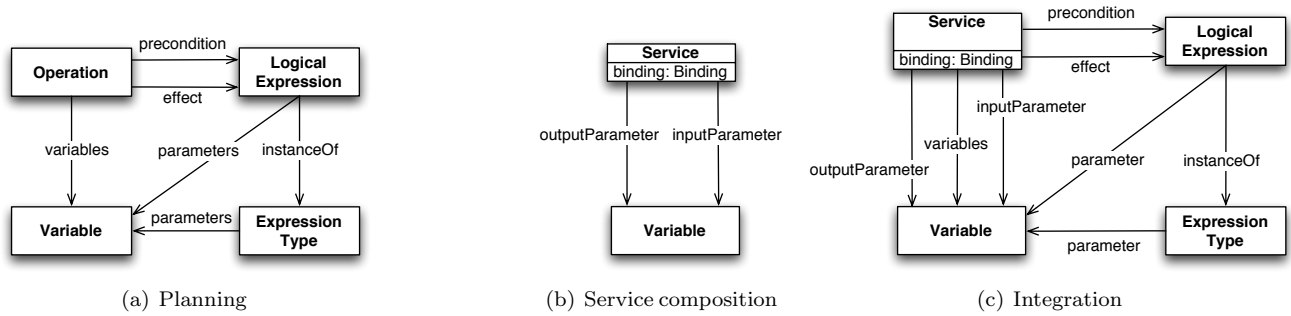


Figure 1. Elements of Composition

2. Integrated Meta-Model

A meta-model defines the concepts and relations in a specific domain. The meta-model for planning for example defines concepts like plans and semantically annotated actions. A service composer that is implemented using planning algorithms actually works on two different meta-models: the meta-model of planning and the meta-model of service composition. The meta-model for service compositions defines the concepts of services and compositions of services. It makes sense to integrate both meta-models for an automated service composer. Additionally this integration gives the service enactor additional information about the service composition. As services are semantically annotated and the enactor knows the original service request by the user, it is able to react sensibly to service failures [20, 6]. It *knows* which service results are important to achieve the overall goal and only terminates enactment if the goal can no longer be reached.

Planning	Service Composition
Plan	Composed Service
Operator	(Atomic) Service
Action	Service Binding
Operator variables	Input- & Output-parameters

Table 1. Mapping from planning to service composition

The integrated meta-model provided in this paper is based on the meta-model of automated planning from Artificial Intelligence [8] and the meta-model of process-based service enactment [2]. Both share a lot of similarities. Table 1 gives an overview of the similar elements. In automated planning the composition is called *plan*, in service composition it is called *composed*

service. The elements of composition are respectively *operators* in planning and *services* or *atomic services* in service composition. To actually invoke an operator or service the formal parameters must be substituted with concrete parameters called *actions* or *service bindings*. During the integration of both meta-models several challenges occurred. For example, planning algorithms generate partially-ordered sets of actions. Current service composition languages like WS-BPEL [15] on the other hand often incorporate block-structured composition mechanisms. The main challenges and our solutions for them are as follows:

Challenge 1: Elements of Composition The elements of composition are the building blocks from which the composition is built up. While they are called operators in planning, they are services in service composition. A service has a name and input and output parameters (Figure 1(b)). In order to allow automated planning the specifications of operators are more complex. It is not sufficient to specify the elements of composition just by their name and input and output parameters. It is necessary to specify the functionality through semantical annotations, too. To do this operators in planning are specified with their preconditions and effects (Figure 1(a)). The former define the constraints that must hold in order to execute the operator; the latter define the impact that the execution of the operator has on the current state. Both – precondition and effect – are specified using logical expressions from function-free first-order-logic. Together with further restrictions this ensures decidability of planning problems [4]. In contrast to the operators from planning that stand for abstract activities, the services symbolize concrete functionality that must be executed. It is therefore necessary to bind the services to concrete protocols and endpoints (e.g. a concrete Web service). This is defined by the *binding* attribute of the service.

To integrate both meta-models, the differences must be merged and conflicts resolved. Figure 1(c) shows

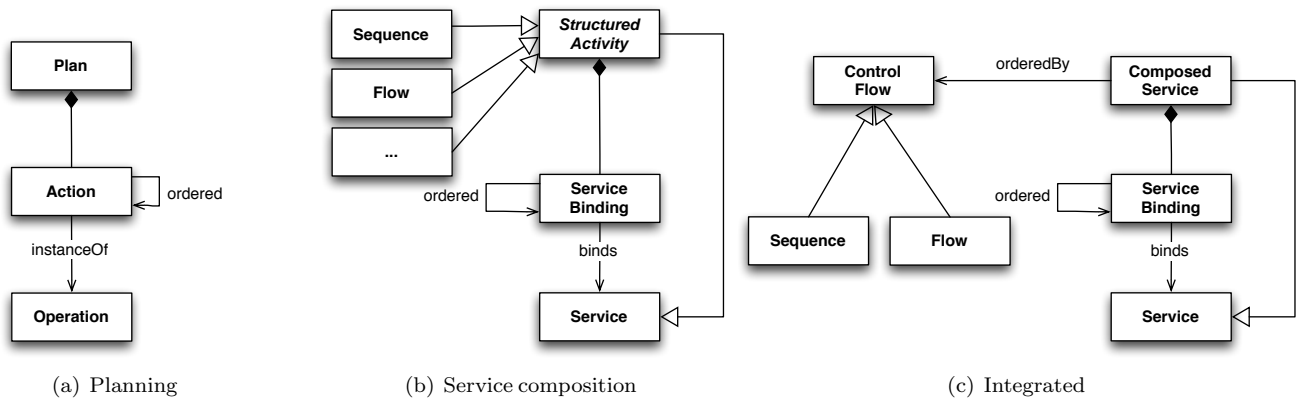


Figure 2. Composition Model

the result of this integration. The elements of composition are *services* and they have a *binding* attribute. To automate service composition the function of the service must be specified using preconditions and effects. Therefore, the approach from automated planning to specify them using logical expressions is adopted. A conflict arises from the difference between operator variables in automated planning and input and output parameters for services. Operator variables are used to declare all variables that are used for defining the operator's precondition and effect. Hence, they incorporate input and output parameters. But they may also contain additional variables that are just needed to define preconditions and effects. To allow these additional variables in our meta-model we could either merge them together with input and output parameters into *variables* or we could have individual attributes for each one. The first approach is the one used in planning. The latter approach is advantageous when services are actually enacted, because it is necessary to separate input parameters and output parameters to allow for a correct invocation. Performing this separation afterwards is very difficult. It therefore makes sense to separate input parameters, output parameters and additional variables of services in our meta-model.

Challenge 2: Composition Model The composition model defines the order in which the elements of composition are arranged. While composition is an integral part of both meta-models, their realization differs. Classical planning only supports the composition of atomic *actions* into plans. Composing a plan out of existing plans is not supported. The composition model of service composition on the other hand supports this *nesting* of one composition into another one. This composition model is similar to

the *Composite pattern* [7] from object-oriented design. The composite – here the *ComposedService* – inherits from the element of composition. Like atomic services, composed services are elements of composition and can be used in other compositions.

The ordering of services inside a composition differs between planning and service enactment, too. In planning the plan is a partially- or totally-ordered¹ set of actions (Figure 2(a)). The ordering is defined by ordering constraints between two actions. Ordering constraints are imposed by causal links and the protection of causal links. An ordering constraint between two actions a_1 and a_2 must be inserted in the following cases:

- a_1 creates a precondition of a_2
- a_1 destroys an effect of a_2 that is protected by causal link
- a_2 destroys a precondition of a_2

The first case is actually a causal link. The second and third cases are called *clobbering* as one activity threatens a causal link. Demotion or promotion must be used to order the activity before or after the causal link. WS-BPEL [15] is used, for the composition of Web Services. It combines the graph-based approach from WSFL [11] with the algebraic or block-structured approach of XLANG [23]. The former is quite similar to the partially-ordered set of actions from automated planning. The latter introduces so called *structured activities*. These are for example *Sequence* (sequential execution of the contained services) or *Flow* (parallel execution). The meta-model of WS-BPEL that combines these two approaches is displayed in Figure 2(b). Composition can be performed through the nesting of structured activities. Additionally it is possible to define or-

¹ Depending on the actual algorithm used.

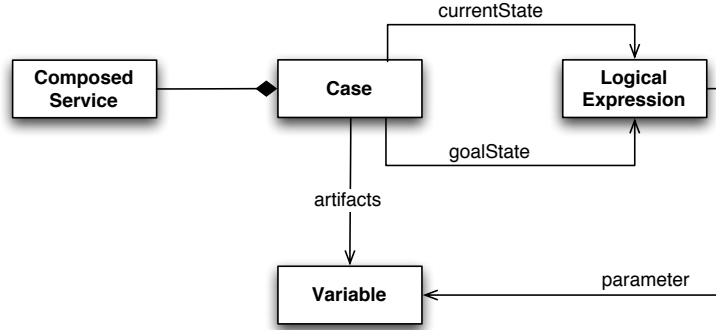


Figure 3. Case-orientation

derings between services. This represents the ability of WS-BPEL to model service compositions in a graph-oriented way.

The first step towards the integrated meta-model displayed in Figure 2(c) is to find a suitable name for the composition. In automated planning it is the *plan* and in WS-BPEL it is a *structured activity*. Both names are not quite suitable. The name *plan* merely represents how it is composed and not what is composed. *Structured activity* does not capture that we are talking about compositions of services. We therefore choose to introduce a new name *composed service*. A composed service is a composition of services that itself is a service. It can be offered as a service and used inside other compositions of services. The main difference of our integrated meta-model to WS-BPEL is that we extracted the control flow that imposes the order of the services inside a composed service into its own class. Instead of specialized composed services we have control flow types for each ordering type. This separates the functional aspect of the process [3] into *what* is executed from *in which order* they are executed.

Challenge 3: Case-orientation The last challenge when integrating planning and service enactment is case-orientation. Planning is done to fulfill concrete tasks. The input for planning is a planning problem that defines the initial state and the goal state. Both contain concrete data. A composed service on the other hand is an abstraction from the concrete case. This allows its reuse. So, planning is case-oriented instead of service-oriented. If planning is used for automated service composition, the result of service composition is not actually a service. Instead it is a process that describes the order in which services must be enacted to fulfill the goal. The advantage of this approach is that the service composition is exactly tailored not only to the precondition and effect of the case but also to its concrete data. The drawback is that such a service composition cannot be easily reused as a composed services in

other cases.

Figure 3 shows the case-oriented approach currently used in the Plængine system. A case in our meta-model identifies a service request. It therefore consists of a current state, a goal state and a set of artifacts. In the beginning the current state describes the initial state of the world. The goal state defines the properties a state must have to qualify as a goal state. Both – current and goal state – are modeled as logical expressions. This similarity to preconditions and effects, allows easy checking whether preconditions are fulfilled in a given state and eases the application of effects to a state. During service composition the case is augmented with the planned service composition that is able to fulfill the goal of the case. During enactment, the Service Enactor updates the current state of the case as services are executed. After the first service of the composition is executed, the effects of this service are reflected in the current state. This does not change the composition, but only the state of the case.

In this section the three main challenges when integrating automated planning and service composition have been presented. They occurred at the level of the elements of composition, at the level of composition and through the case-orientation of planning. Figure 4 shows the integrated solution for all three presented challenges. A complete version and in-depth discussion of this meta-model is presented in [13] and [16]. Besides these main challenges additional challenges exist that cannot be presented in this paper. These include for example the different data models of planning, that is based on first-order logic, and of services that consists of hierarchically structured data. In the latter a data element *Address* consists of further data elements (e.g. *Street*), while in the previous this is modeled as a relation (e.g. *inStreet(address, street)*).

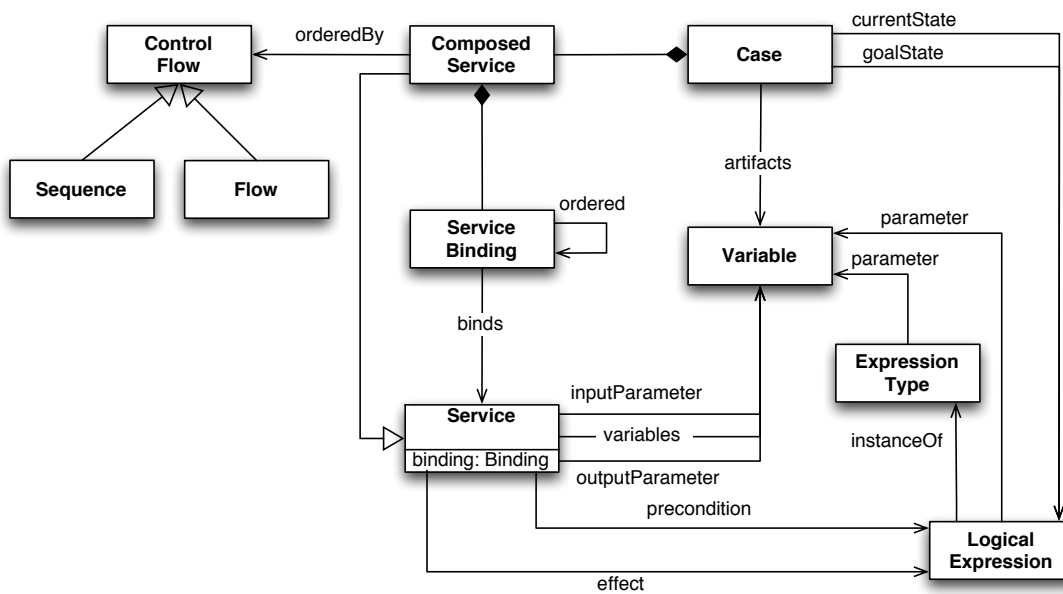


Figure 4. Integrated Conceptual Model

3. Platform Architecture

The Plængine system described in this paper provides a platform for the service-oriented architecture. The basic principles of the service-oriented architecture are shown in Figure 5. Service provider and requester are decoupled, services are published and discovered in registries. The binding of a service is deferred, therefore late-binding is used. Each service request is stateless. This design allows for maximum flexibility, as both the providers as well as services can change at any time and switching between them is eased as much as possible.

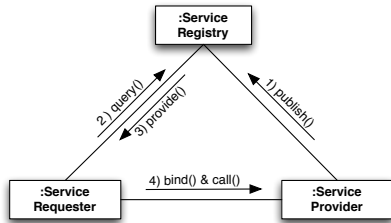


Figure 5. SOA - The big picture

Starting from this general architecture, we will now introduce the Plængine platform architecture as shown in Figure 6. As Plængine is designed to dynamically create and execute composed services tailored to the service request as specified by the user (i.e. service re-

quester), it has to act as a service provider. The services provided by Plængine are dynamically composed. To compose such services, the Plængine system will rely on a composer component. Once composed, the service composition is executed by the enactor component.

In the service-oriented architecture the enactor is a service requester itself, as it will call other services during execution. Compared to traditional systems, the enactor component is equivalent to a workflow engine. All components require the domain registry, storing the service specifications. These specifications need to be semantically enriched, to allow for automatic composition. The domain registry itself will interact with external service registries to acquire additional services. In the long run, the domain registry can replace the external service registry, if all parties agree on a common semantic description of services.

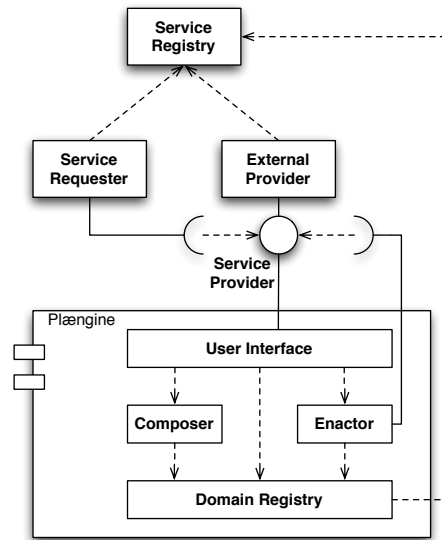


Figure 6. Plængine architecture

The strength of the Plængine system is that the general structure of the service-oriented architecture is not altered, but augmented. Plængine seamlessly integrates into an existing service-oriented architecture and helps to produce added value by composing new services on top the existing ones.

A typical use case of the Plængine system is a market place. As opposed to production workflows, service compositions are short-lived (e.g. there are only 300 seats on a plane to sell) and the requests are very specific (e.g. a journey is restricted by time, location, and budget). Figure 7 (p. 7) shows a typical use case se-

quence of the Plængine system. The service requester sends a request containing current state and goal state to the Plængine’s user interface. The user interface will create a *Case* containing current and goal state. This case is sent to the composer to create a process representing the service composition requested. To do so, it will rely on the domain registry to provide semantically enriched service specifications. If successful, the composition is stored within the case and the user interface is notified. Failed planning is caused by missing services to reach the goal state. This is unlikely, as we look at specific domains. Next, the user interface will send the case to the enactor. The enactor uses the service composition stored inside the case to locate the individual services of the composition via the domain registry and invoke them. On successful completion, Plængine’s user interface is notified and can then provide a service response to the service requester. If the invocation of services within the composition fails, the design of the Plængine system allows for replanning by transferring the case back to the composer. This is another benefit of the integrated meta-model presented in this paper.

In the following two subsections, we will look at the architecture of the composer and of the enactor in more detail, based on the example use case sequence just given.

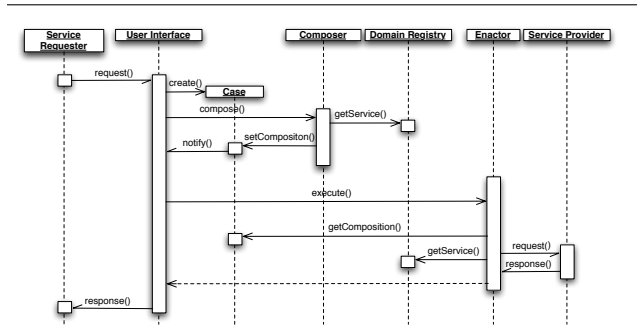


Figure 7. Example Plængine interaction

3.1. Composer Architecture

Figure 8 shows how the composer of Plængine works internally. After the case has been created by the user interface it is delivered to the composer. Instead of performing the composition itself the composer delegates this task to another component *ComposerThread*. The *ComposerThread* uses a heuristic search planning algorithm to perform service composition. Search takes

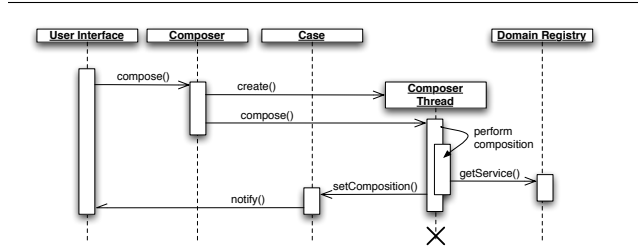


Figure 8. Decomposition of the Plængine composer

place in state space [8] and is directed forward. This means that the search starts at the current state and tries to find a path from this state to a goal state. The heuristic is used to guide the search by estimating the distance of states to a goal state. The algorithm used is an enhancement to the one presented in [14]. It supports the composition of services with parallel and alternative control flows. It does so by maintaining not one current state but a set of possible current states and simultaneously selecting multiple services for integration in the composed service. The exact algorithm used by the composer is introduced in [13].

3.2. Enactor Architecture

Traditionally, workflow patterns [10] have been used to evaluate and compare workflow languages. They provide evaluation criteria as well as a means of reducing the complexity of such an evaluation. Some systems or languages, such as YAWL [24], are specifically designed to implement as many workflow patterns as possible. Plængine takes this idea one step further, by using workflow patterns as the fundamental concept for the design of a process enactment engine. This leads to an explicit modeling of control flows, as shown in Figure 2(c). The scope of such an explicit control flow is local to a composed service, i.e. it can only order the execution within one composed service. Most workflow patterns can directly be mapped to a control flow instance, reducing the construction to pick and choose.

In the following, the architecture of the enactor is explained in more detail. The enactor itself is composed of two main components, the *EnactorImpl* and the *Task Scheduler*. The former is responsible for enforcing the enactor’s contract by coordinating the various components. The Task Scheduler is responsible for thread management and execution of the service implementation. For Web Services, the implementation provides send-, receive-, and reply-methods for the actual services.

As mentioned above, a composed service has an explicit control flow responsible for scheduling the contained services in the correct order. When a control flow instance schedules the execution of a service instance, the `EnactorImpl` is notified of the new state. It will look up the service's specification and create a service implementation instance. The created service implementation is then passed to the Task Scheduler, which will eventually cause a thread to execute.

Once executed, the service instance is set to *executed*, which will cause a notification to the `EnactorImpl`. The `EnactorImpl` will notify the assigned control flow and the cycle starts again.

In reality, the `EnactorImpl` is more fine grained, with one specific instance for each type of notification. Thus, the complete behavior of the enactor can be changed by simply rewiring the notification chain. This design provides maximum separation of concerns and flexibility.

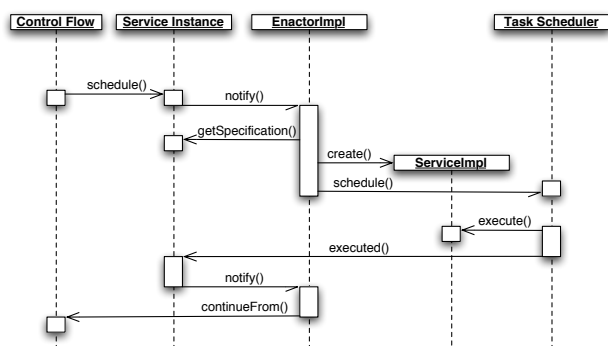


Figure 9. Decomposition of the Plø engine enactor

Yet, there are workflow patterns crossing the boundaries of composed services, e.g. the *Milestone Pattern*. Figure 10 shows a very simple process that can not be constructed with the meta-model outlined so far.

WS-BPEL [15] provides a solution to this problem via *Links*, implementing directed edges across blocks while observing hierarchy restrictions. Instead of just copying this feature, we introduce a generalized concept we call *service life-cycle interceptors*. Service life-cycle interceptors can be attached to a service binding (see Section 2) as listeners to life-cycle events. The following life-cycle events are observable:

- *Readiness* is the point in the service's life-cycle, when it's readiness for execution is evaluated, i.e.

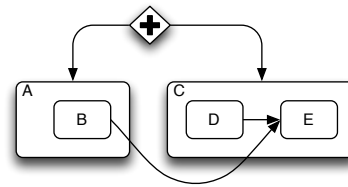


Figure 10. Example control flow crossing boundaries

the assigned control flow has scheduled the service's execution. A *Readiness* interceptor can react to the imminent execution or defer it. Thus, the equivalent of a WS-BPEL Link Target can be implemented.

- *PostExecute* is the point in the service life-cycle, when its execution has just completed, regardless of success. This is the foundation for the equivalent to a WS-BPEL Link Source.
- *Exception* and *Compensation*. These life-cycle events provide the necessary hooks for exception handling and compensation. Yet, their semantics differ slightly from the former, as exception and compensation handlers are chained hierarchically. If a handler is unable to provide a suitable handling or there is no handler at all, the service's parent (e.g. a nested composed service) is searched for a suitable handler.

Figure 11 shows how the simple example from Figure 10 is implemented by service life-cycle interceptors. Only if both the control flow and the *Readiness* interceptor schedule the service *E* to execute the enactor will do so. Yet, the *Readiness* interceptor of *E* will only schedule after notification from the *PostExecute* interceptor of *B*. This effectively implements a directed edge (i.e. causal link) between the two services. In [16] it is shown that all workflow patterns known today, can be easily implemented with the given meta-model. The availability of service life-cycle interceptors in the meta-model provides openness to future extensions, possibly advanced workflow patterns.

4. Related Work

Several approaches for automated service composition [18, 21, 25] or service enactment exist, but only few integrate both. The Web Service Execution Environment (<http://www.wsmo.org/wsmx/>) is the approach to develop an execution environment for Seman-

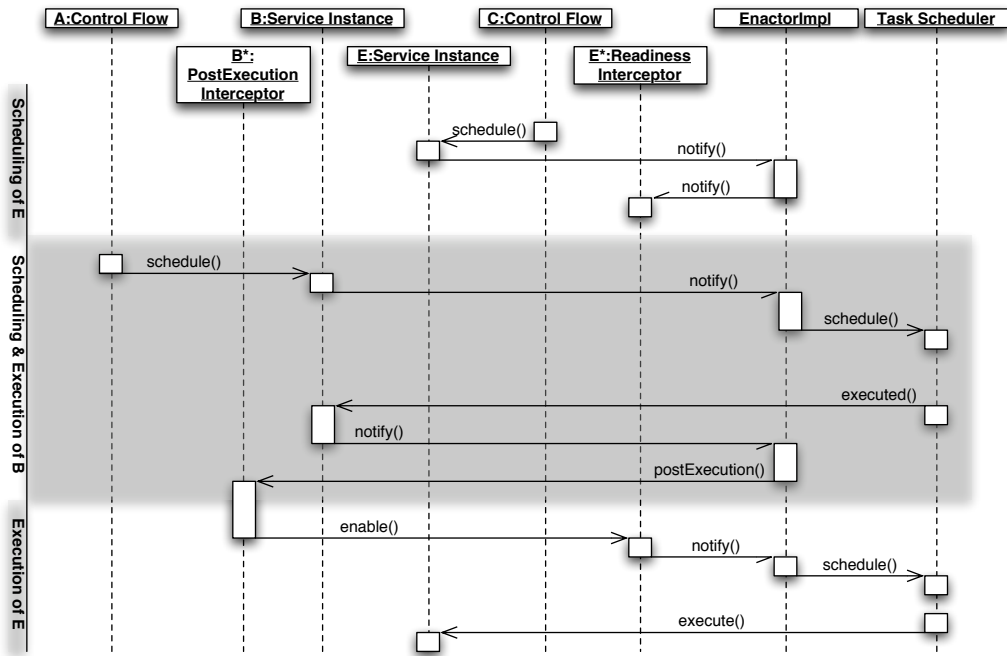


Figure 11. Implementation of service life-cycle interceptors

tic Web Services with the main focus on automated service discovery and selection. It is based on the Web Service Modeling Ontology and the Web Service Modeling Language that provide a meta-model quite similar to the one presented in this paper. The main difference is the focus on service discovery and selection instead of on automated service composition². Like WSMO/L/X, OWL-S [17] does provide a meta-model similar to the one we presented in Section 2. Meteor-S [22] is a pragmatic approach to extend Web Service standards with semantic web technologies. WSDL-S, for example, adds semantic annotations to web service operations specified in WSDL. Meteor-S itself provides components to perform service composition, service discovery, late service binding and service enactment.

The results of the Plængine project are currently used and extended in the European research project Adaptive Services Grid (<http://asg-platform.org>). It aims at developing a platform for highly dynamic environments like market places or telematics. It will allow the on-demand composition and creation of services. Quality of service constraints may be negotiated with service providers and are monitored during service enactment. Services are executed on a grid that forms a homogenous execution environment from which resources may be removed or to which resources may be

added.

5. Conclusion

In this paper design and implementation concepts for Plængine, a system for integrated service composition and enactment is presented. While the overall architecture and system components are expected stable in future versions, the service composition algorithms used in Plængine might change to incorporate new research results.

An aspect not covered in this paper is the handling of unexpected events during enactment. Plængine does not use static exception handling mechanisms to cope with them but uses re-planning [20, 6] instead. If for example an error occurs because a service delivers an unexpected result, enactment is halted and a new, changed composition is created to handle the user request starting from the current state.

Today's planning algorithms generate partially-ordered sets of services. The translation of these ordered sets into more advanced control flow structures (like those defined in workflow patterns), is also an area of future work. Another aspect for future work is an external (e.g. visual) representation of service compositions. Currently, a control flow is represented within the system by a definition and an implementation. Displaying such a

² Though service composition will be supported in the future.

construct in any other format requires more information, currently not stored within the meta-model.

Acknowledgments The authors would like to thank Jens Hündling and Hilmar Schuschel for their involvement in the Plængine project and their valuable suggestions on earlier versions of this paper.

References

- [1] A. Brown, S. Johnston, and K. Kelly. Using Service-Oriented Architecture and Component-Based Development to Build Web Service Applications. Technical report, IBM (Rational Software Corporation), 2002.
- [2] F. Casati, H. Kuno, G. Alonso, and V. Machiraju. *Web Services - Concepts, Architectures and Applications*. Springer, Heidelberg, November 2003.
- [3] B. Curtis, M. I. Keller, and J. Over. Process Modeling. *Communications of the ACM*, 35:75–90, 1992.
- [4] K. Erol, D. S. Nau, and V. Subrahmanian. Complexity, decidability and undecidability results for domain-independent planning: A detailed analysis. Technical Report CS-TR-2797, UMIACS-TR-91-154, SRC-TR-91-96, University Of Maryland, 1991.
- [5] R. Fikes and N. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.
- [6] M. Gajewski, H. Meyer, M. Momotko, H. Schuschel, and M. Weske. Dynamic failure recovery of generated workflows. In *Proceedings of the 16th International Conference and Workshop on Database and Expert Systems Applications*. IEEE Computer Society Press, 2005. (to appear).
- [7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley Professional, Reading, Massachusetts, January 1995.
- [8] M. Ghallab, D. Lau, and P. Traverso. *Automated Planning - theory and practice*. Morgan Kaufmann, Amsterdam, 2004.
- [9] K. v. Hee and W. v. d. Aalst. *Workflow Management: Models, Methods, and Systems (Cooperative Information Systems)*. The MIT Press, Cambridge, Massachusetts, January 2002.
- [10] B. Kiepuszewski, A. Barros, W. v. d. Aalst, and A. t. Hofstede. Workflow patterns. *Distributed and Parallel Databases*, 14:5–51, 2003.
- [11] F. Leymann. *Web Services Flow Language (WSFL 1.0)*. IBM Corp., 2001.
- [12] F. Leymann and D. Roller. *Production Workflow: Concepts and Techniques*. Prentice Hall, Upper Saddle River, New Jersey, 2000.
- [13] H. Meyer. Development and Realization of a Planning Component for Service Composition (in German). Diploma Thesis, Institute of Computer Science, University of Potsdam, 2005.
- [14] B. Nebel and J. Hoffmann. The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.
- [15] Organization for the Advancement of Structured Information Standards. *Web Services Business Process Execution Language (WS-BPEL)*, 2004. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel.
- [16] H. Overdick. Design and Implementation of Process Enactment Components in Service-Oriented Environments (in German). Master’s thesis, Hasso-Plattner-Institute for IT Systems Engineering, University of Potsdam, 2005.
- [17] OWL Service Coalition. *OWL-S 1.1*, 2004.
- [18] M. Pistore, F. Barbon, P. Bertoli, D. Shaparau, and P. Traverso. Planning and monitoring web service composition. In *Workshop on Planning and Scheduling for Web and Grid Services (held in conjunction with The 14th International Conference on Automated Planning and Scheduling)*, pages 70–71, 2004.
- [19] H. Schuschel and M. Weske. Integrated Workflow Planning and Coordination. In *Proceedings of 14th International Conference on Database and Expert Systems Applications*, volume 2736 of *Lecture Notes in Computer Science*, pages 771–781. Springer, Heidelberg, 2003.
- [20] H. Schuschel and M. Weske. Triggering Replanning in an Integrated Workflow Planning and Enactment System. In *Proceedings of 8th East-European Conference on Advances in Databases and Information Systems*, volume 3255 of *Lecture Notes in Computer Science*, pages 322–335. Springer, Heidelberg, 2004.
- [21] E. Sirin, B. Parsia, D. Wu, J. Hendler, and D. Nau. Htn planning for web service composition. *Journal of Web Semantics*, 1(4):377–396, 2004.
- [22] K. Sivashanmugam. The METEOR-S Framework for Semantic Web Process Composition. Master’s thesis, Department of Computer Science, University of Georgia, 2003.
- [23] S. Thatte. *XLANG*, 2001.
- [24] W. van der Aalst, L. Aldred, M. Dumas, and A. ter Hofstede. Design and Implementation of the YAWL system. In *Proceedings of 16th International Conference on Advanced Information Systems Engineering*, volume 3084 of *Lecture Notes in Computer Science*, pages 142–159, Heidelberg, 2004. Springer.
- [25] L. Zeng, B. Benatallah, H. Lei, A. Ngu, D. Flaxer, and H. Chang. Flexible Composition of Enterprise Web Services. *Electronic Markets – Web Services*, 13:141–152, 2003.