Distributed and Parallel Databases manuscript No.
(will be inserted by the editor)

SEPL – A Domain-Specific Language and Execution Environment for Protocols of Stateful Web Services

Waldemar Hummer · Philipp Leitner · Schahram Dustdar

Received: date / Accepted: date

Abstract In order to interact with a stateful Web service, clients need to obtain information about the intra-service protocol, which contains valid operation sequences and the expected input-output transformation across invocations. While the community has widely agreed on WSDL as the standard for functional service description (the "static" service interface), there is still an evident lack of languages to describe the dynamic, behavioral interface of services. In this paper we introduce *SEPL* (*SErvice Protocol Language*), a domain-specific language (DSL) for defining executable intra-service protocols. Notable features of the DSL include support for WS-Addressing and simple creation of new Web service instances, synchronous and asynchronous service invocation facilities and easy access to WSRF-style service resource properties. Service providers use SEPL to define the procedure that clients must adhere to in order to achieve a certain higher-level functionality. Clients use the combined information of the SEPL document and the WSDL definitions to execute an intra-service protocol. We provide a graphical representation of SEPL the form of UML Activity Diagrams, and tools to generate executable code from these models. We further present a solution to host and execute SEPL protocols in a server application based on Web services technology.

Keywords Web Services · Intra-Service Protocol · Stateful Web Services · Domain-Specific Language · Service Protocol Language · SEPL

Waldemar Hummer (✉) · Philipp Leitner · Schahram Dustdar
Distributed Systems Group
Vienna University of Technology
Argentinierstrasse 8/184-1, 1040 Vienna, Austria
E-mail: hummer@infosys.tuwien.ac.at

Philipp Leitner
E-mail: leitner@infosys.tuwien.ac.at

Schahram Dustdar
E-mail: dustdar@infosys.tuwien.ac.at

1 Introduction

Throughout the last years, software engineering research and practice have put remarkable focus on the Service-Oriented Architecture (SOA) [29,11,22] paradigm, which propagates the use of services as a means to create decoupled, distributed, composite applications in heterogeneous environments. Services are autonomous applications made available in a computer network using standardized interface description and message exchange. *Web services* have gained momentum as a means for implementing SOA applications and services. It is a commonly agreed principle that Web services generally do not persist a state across invocations, i.e., that they are stateless [11,22]. However, in some areas stateful services have indeed become a necessity. Most notable is the concept of the Grid service which is defined in [14] as follows: *"a Web service [...] that implements standard interfaces, behaviors, and conventions that collectively allow for services that can be transient (i.e., can be created and destroyed) and stateful (i.e., we can distinguish one service instance from another)"*. Additionally, distributed data management and integration architectures (e.g., [5,16]) often also rely on stateful services. Furthermore, *Data-as-a-Service* [38] approaches use Web services to provide data on demand, following certain access control models and query protocols. The *Web Services Resource Framework* (WSRF) [27] builds a foundation for creating, addressing and destroying service resources and for accessing the data (or properties) exposed by these resources.

In general, clients need to obtain information about a service in order to access its exposed data, or to be able to successfully invoke one or more of the service's operations. On the one hand, this information concerns the "static" interface description, including the names of available operations, parameter and return types as well as the message style to be used. These issues are well covered by the WSDL contract offered by the service provider. On the other hand, for stateful services, clients additionally ought to know the service's "dynamic" (behavioral) interface, which specifies the order in which operations can be invoked and in which way the data of a service may be retrieved and modified. As illustrated in

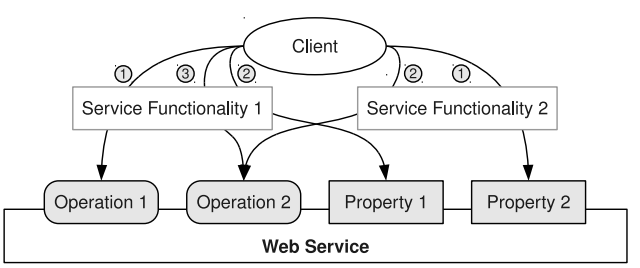

Fig. 1 Service Functionalities

Figure 1, Web services often provide different functionalities which may involve invocation of more than one service operation and access to several data properties. This especially applies to stateful services, i.e., services which persist data values across invocations. We refer to such constraints, which require the client to have knowledge about functionalities on top of the actual service operations,

as the *intra-service protocol* (for short, service protocol). We observe that service protocols have transactional characteristics: if one involved operation call fails, the wanted functionality can most likely not be delivered.

Service protocols play a key role for transactional services [31, 21] and distributed databases. A transactional service embraces a sequence of actions that must be executed as a unit [21], similar to the service functionalities of stateful services. In the Web service environment, SOAP (Simple Object Access Protocol) Faults are used to indicate exceptional and erroneous behavior. When an error is detected, some compensation routine can be started to rollback the performed changes and to reconstruct a consistent state. The Web services standards for defining distributed transactions are 1) WS-AtomicTransaction (AT) for short-running transactions with an all-or-nothing property, and 2) WS-BusinessActivity (BA) to handle long-lived activities and business protocols. Both specifications build on WS-Coordination, which defines a coordinator service that is used by the participating services to activate a coordination context, to register for a certain transaction protocol and to execute it. AT follows a traditional approach from the database domain and enforces the ACID (atomicity, consistency, isolation, and durability) criterion using a two-phase commit protocol. These ACID transactions are usually not applicable for long-running activities because resources cannot be locked in a transaction that runs over a long time. Therefore, BA defines compensating transactions, which provide a means to undo an action if a process or user cancels it [31]. In any case, AT and BA require a centralized coordinator entity.

1.1 Motivation

Intra-service protocols have commonalities with transaction protocols in distributed databases and combine properties from both AT and BA. A sequence of actions (invocations) needs to be executed, and the result of the sequence depends on the proper execution of all atomic actions. However, the fact that only one service participates in a protocol functionality renders the coordinator service's functionality for activation and registration unnecessary. Hence, coordination and execution of an intra-service protocol is performed solely between client and target service.

The description of intra-service protocols is essentially a subproblem of service composition [9]. Therefore, languages from the service composition domain (e.g., WS-BPEL [26]) can also be used to specify intra-service protocols. However, this approach has a number of drawbacks. Firstly, we argue that the problem of intra-service protocol specification is considerably less complex than service composition. The service to invoke is always clearly defined (e.g., there is no need for different partner link types), and protocols are usually much less complicated than compositions. The related work discussion in Section 7 compares in more detail which language constructs are (not) supported in intra-service protocols as opposed to WS-BPEL. Secondly, composition languages are generic and do not contain any explicit support for stateful Web services specifics, such as *Resource Properties* [28] in WSRF or patterns for resource lifetime. Thirdly, composition engines are rather heavy-weight server tools, and not suitable for client-side usage. Lastly, the XML syntax of languages such as WS-BPEL is notoriously hard to write without appropriate tool support. For a light-weight Domain-Specific Language [34] (DSL) a simpler and easier-to-understand syntax may therefore be superior.

Currently, there is still an evident lack of special purpose languages to describe executable intra-service protocols for stateful Web services. The discussion of related work in Section 7 outlines that most existing approaches and standards for intra-service protocols are rather a high-level guideline for clients than an executable language. We tackle this particular problem and present a domain-specific language and an execution environment for protocols of stateful Web services.

1.2 Example Scenario

As the motivating example we consider a service hosted by a European cell phone operator (CPO). The service allows other CPOs to port customer telephone numbers from one provider to the other, a functionality that CPOs have to provide because of European Union regulations. The service is implemented as a stateful Web service using WSRF, and employs the factory/instance pattern (one stateless factory service is used to create stateful Web service resources).

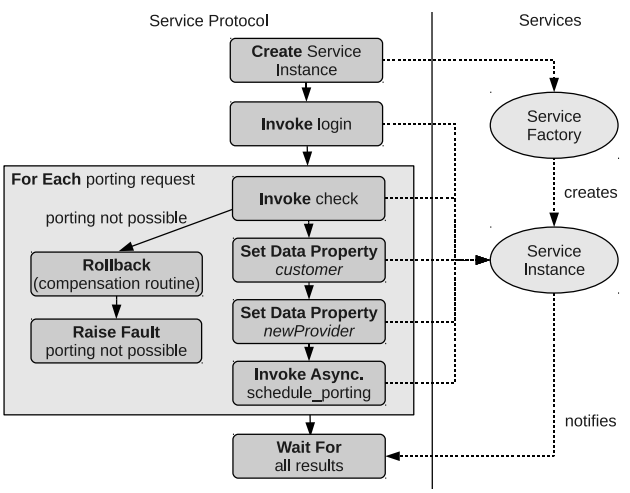

Fig. 2 Intra-Service Protocol for Number Porting Functionality

Figure 2 sketches the intra-service protocol for the number porting functionality in a simple graphical notation. The left-hand side of the figure depicts the steps to be carried out by the client, while the right-hand side shows the services involved. Firstly, the client has to create a new `Number Porting` service instance by using the `Service Factory`'s `create` operation. This operation returns a reference to the new instance, which is used by the client in all subsequent invocations. After being granted access to the actual service functionality by logging in, the client has to loop over all porting requests and check whether the porting process is possible. If it is not, the client interrupts the current execution and calls a rollback routine. If porting is possible, the client provides the necessary input to the service by setting two WSRF resource properties (`customer` and `newProvider`). Then the porting can be scheduled for a specific date, causing the porting operation to be carried out asynchronously. The service returns the result of this

operation by sending a notification to the client. The protocol ends at the point where all notification results have arrived at the client.

Even though this example is simple to understand, it contains a number of challenges and issues that service providers may encounter when defining intra-service protocols, including invocations of Web service operations, alternative branches of execution, service callbacks, (SOAP) fault handling, and handling of WSRF resource properties. Furthermore, the number porting protocol has a transactional aspect, since all requested portings passed as input to the protocol need to be completed, and the operation is rolled back in case one request cannot be completed. It should also be noted that the number porting scenario defines a rather intensive business logic, and real-world protocols may be significantly simpler. The scenario has been chosen to illustrate the majority of the SEPL features in one example.

1.3 Contribution

We address the issues mentioned above and propose a framework for intra-service protocol description, modeling, and execution. The contribution is threefold:

- We introduce a light-weight DSL named *SEPL* (*SErvice Protocol Language*), which offers features to specify functionalities on top of the operations of a Web service. Features of SEPL include synchronous and asynchronous invocations, fault handling, simplified processing of XML messages and direct support for WSRF specifics and for the service factory/instance pattern. An advantage of SEPL documents is that they are decoupled from existing services and that service implementations remain untouched. SEPL documents are straight-forward to author for service providers and easy and efficient to interpret for clients.
- Following a model-driven approach [1], SEPL protocols can be modeled as UML (Unified Modeling Language) activity diagrams [23]. Existing UML tools facilitate the development process and help to graphically compose intra-service protocol activity diagrams (PADs). A PAD definition provides the required information to generate executable code in the syntax of a scripting language. We define a 1-to-1 mapping from UML activity diagrams to executable SEPL code, and present the implementation of the SEPL code generation tool *UML2SEPL*.
- We present a prototype SEPL execution client written in the Java programming language. The SEPL client combines the information from WSDL and SEPL documents and conducts the SOAP message exchange as mandated by the service protocol. Additionally, to take away the responsibility of clients to execute SEPL protocols, we offer a SEPL protocol server implementation. The task of the server is to host SEPL protocols, to expose functionalities contained therein as WSDL operations and to execute protocol functions upon request.

The remainder of this paper is structured as follows. Based on the motivating example scenario, we introduce the UML notation of SEPL and define the concepts and language features in Section 2. Section 3 presents the and describe the mapping of UML to executable SEPL code. Section 4 describes how intra-service protocols can be centrally hosted in a server application. Section 5 discusses the implementation of the SEPL client and the protocol host. An evaluation of the SEPL framework is carried out in Section 6. Section 7 discusses existing work related to our approach and the paper concludes with a brief summary in Section 8.

2 SEPL – The Service Protocol Language

In this section we present the concepts employed by SEPL and describe the syntax and purpose of its concrete language constructs. Development of SEPL protocols follows a model-driven approach, based on UML activity diagrams. The presentation of the language details is based on the UML representation of the sample scenario, which is depicted in Figure 3. All SEPL-specific language constructs in the figure are printed in **bold**, whereas scenario-specific parts are printed in *italics* (names of service operations to invoke) or in normal font (variables, constants and data properties). In the following, a SEPL protocol in UML notation is referred to as *service protocol activity diagram*, or *PAD*.

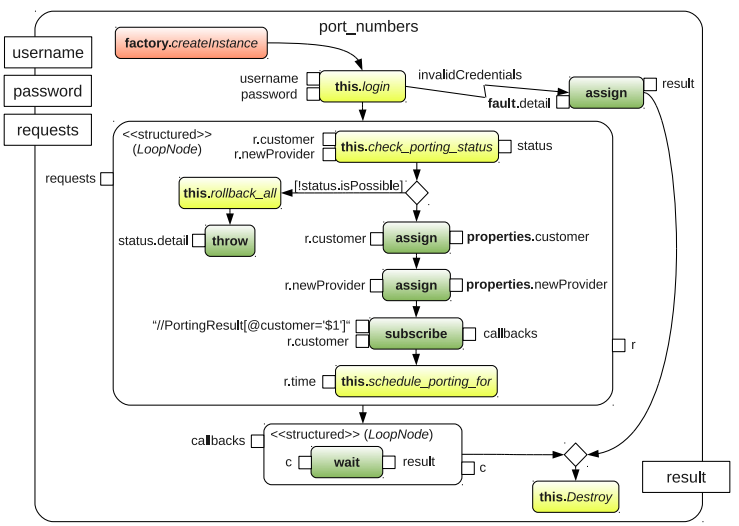

Fig. 3 UML Activity Diagram of Sample Service Protocol

2.1 Scenario Service Protocol

The top-level structure in SEPL documents, the protocol *function*, is modeled as a UML activity. The scenario protocol function `port_numbers` is depicted in Figure 3. The activity defines three input pins, which constitute the input parameters needed by the service protocol: (`username`, `password` and `requests`). Parameters are immutable and cannot be changed inside an activity. The activity result is defined using an output parameter, in the example using the variable named `result`. The UML standard specifies that an activity contains executable nodes and control nodes as well as edges between these nodes. The node types are further divided into different subtypes. Each edge connects exactly two nodes and the directions of the edges signify the control flow of the service protocol. Note that PADs contain no UML *object flow* edges but only *control flow* edges. The `port_numbers` activity contains both structured activities (in the form of two loop nodes) as well as atomic

actions. Actions are used for resource creation/destruction, synchronous service invocations, fault handling statements and directives for asynchronous invocations. In the figure, input pins are depicted on the left side and output pins on the right side of an action. We define that each input pin is an *action input pin*, whose *fromAction* association points to a *value specification action*, which itself references and evaluates the value of the input pin. The advantage of this approach is that input pins can hold 1) constant values (e.g., `"//PortingResult..."`), 2) variable references (e.g., `username`) and 3) expressions to be evaluated (e.g., `r.customer`). In Figure 3 the input pin values are printed next to the input pin, regardless of which of the three types the input has. The input and output pins of the loop nodes are the UML-defined *loopVariableInput* and *loopVariable*, respectively. For simplicity, the *bodyOutput* pins are omitted; we define that in each iteration of the loop the next array element of the *loopVariableInput* is put to *loopVariable*.

SEPL PADs allow for the use of *variables* as defined in the UML superstructure specification. All *structured activities* (e.g., activities, loop nodes) in the diagram may hold references to variables that are local to the scope of this activity (indicated using the *activityScope* association of the variable). A variable that holds no *activityScope* association is a global variable. All data dependencies between the actions in a SEPL protocol are modeled using variables, and object flow edges are not required. Variables are untyped (i.e., the UML *type* association is not set) and the type of a variable and the operations supported by the underlying object are determined by the SEPL interpreter at runtime (discussed later in Section 5). Variables can hold basic types (e.g., numbers, strings), arrays, object references and XML structures, which are treated specially in SEPL (see Section 2.3). The assignment of values to variables uses the UML *AddVariableValueAction*. In the sample this happens either explicitly with the actions named `assign` or implicitly when assigning the result of any other action via an output pin. If the attribute *isReplaceAll* is set to `true`, the target variable is overwritten; otherwise the variable is an array and the value gets appended to the array end. Note that this is not visible in the graphical representation but contained in the metadata of the UML elements. The example uses three array variables: `requests` is the activity input and contains the porting requests, `callbacks` is used for the output pin of the `subscribe` action, and `result` contains all individual number porting results.

Table 1 Reserved SEPL Variables

Name	Purpose
`async`	Interface for Asynchronous Invocations
`factory`	Reference to the Factory Service
`fault`	Reference to a Caught SOAP Fault
`properties`	Container for WSRF Resource Properties
`this`	Reference to the Target Service Instance

Five globally defined variable names (`async,factory,fault,properties,this`) in SEPL are reserved for special purposes and must not be assigned values (see Table 1). The semantics of these language constructs are explained in the following subsections.

The invocation of a Web service operation in SEPL PADs is modeled using a *CallOperationAction*. The associated *target* is either `this` (to invoke the target Web service this protocol applies to) or `factory` (to invoke the service factory). For illustrative purposes the *target* is printed in front of the operation name in Figure 3. An invocation action contains the name of the WSDL operation as well as a list of parameter input pins, and returns a result which can be directly assigned to a variable via an output pin. Figure 4 depicts the relationship between the graphical representation of a SEPL invocation, WSDL and SOAP. From the SEPL PAD description the name of the operation is determined and its definition is looked up in the WSDL. The XML schema of the operation's input element is matched with the parameter input pins of the action to finally construct a SOAP message and send it to the target Web service.

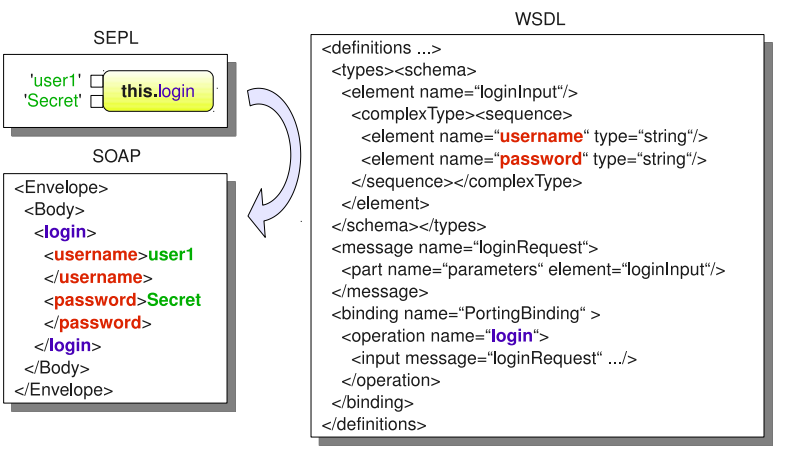

Fig. 4 SEPL-to-SOAP Mapping

To define the behavior of service protocols, SEPL supports a number of standard control flow structures (if-then-else branch, loop node, fault handler). The number porting protocol contains two *loop nodes*, which iterate over all elements of an array input variable (`requests` and `callbacks`, respectively). The example also contains one if-then-else branch with a *guard* expression `!status.isPossible`, which evaluates the output of the `check_porting_status` service invocation. A protocol function can return results (e.g., indicating outputs of the service protocol, returning status codes) using the activity output parameter (`result`). Intra-service protocols are not meant to perform heavy computations, but rather to delegate tasks to existing services and to process their results. Nevertheless, the basic arithmetic, logic, string processing and comparison operations are supported in SEPL.

2.2 WSRF Specific SEPL Features

The WSRF set of specifications [27] defines a message exchange model and related XML definitions to access stateful (computational) resources, which retain a state between invocations. Influenced by the observation that stateful service computing

has gained considerable importance, the design of SEPL is tailored to specifically support concepts of the WSRF.

The WSRF builds on WS-Addressing, a standard to uniquely identify Web services. WS-Addressing introduced the notion of stateful interactions and provides a means to address service instances that are created as the result of these interactions. A service's EndpointReference (EPR) contains information about the location of the service as well as instance-specific configuration details. The WSRF suggests the use of factory services to control the lifetime of service resources (often referred to as the factory/instance pattern). Upon request, the factory creates a new instance of a particular service and returns the EPR which holds the details (identifier) of the new instance. For this purpose, SEPL provides the predefined object named `factory`, which acts as a proxy to the factory service. In the scenario, the factory's operation `createResource` is used to create a new resource (see Figure 3). The invocation response contains the new EPR, which gets assigned to the predefined variable `this.EPR` (not explicitly shown in the UML notation). Finally, the service instance's `Destroy` operation is invoked to destroy the resource.

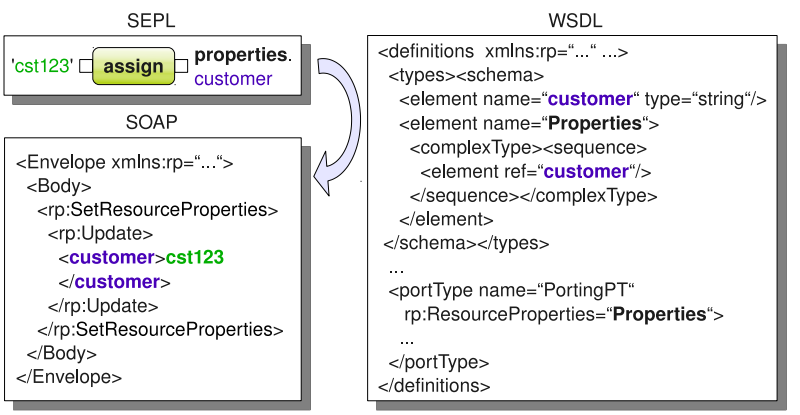

Fig. 5 Setting WS-Resource Properties

This concept of stateful services and resources in WSRF is further extended by the WS-ResourceProperties (WS-RP) specification. In WS-RP, the state of a stateful service resource (instance) is defined by a set of properties, which are exposed in the service description (WSDL) and can be retrieved and updated using standardized operations (`GetResourceProperty`, `SetResourceProperties`). Using the predefined `properties` object, SEPL allows direct read and write access to resource properties. In the protocol in Figure 3 we use resource properties named `customer` and `newProvider` to specify which customer account should be ported to which provider. Figure 5 depicts the relationship of SEPL's UML syntax to WSDL and SOAP, applied to the example for setting the resource property `customer`. The WSDL document of the target service resource contains a definition of the property and its type. The SEPL command is transformed to a service invocation with a `SetResourceProperties` SOAP body element. This method is the standardized way in WSRF to update the value of a resource property.

2.3 Advanced SEPL Concepts

A key goal of SEPL is to simplify access to elements and attributes of XML structures. Assuming a variable `var` contains an XML structure, an XML sub-element named `element` can be directly addressed using `var.element`. Table 2 shows an extended example of how XML elements and attributes can be accessed in SEPL. SEPL's convenience syntax to access XML sub-elements is equal to XPath with the difference that a dot (`.`) is used instead of a slash (`/`). The reason for using this dot syntax is twofold: first, the notation is compliant with the syntax of accessing object attributes in UML's Object Constraint Language (OCL); second, SEPL activity diagrams are intended to be converted into an object-oriented, executable scripting language (see Section 3) and XML markup is internally represented as an object tree. The lookup of object associations is integrated in the scripting engine (see Section 5) and slightly faster than applying XPath. XPath expressions may as well be used directly as can be seen in Table 2. Note that array indices in XPath start at position `1` whereas in SEPL array indices start with `0`.

Table 2 XML Usage in SEPL

Content of Variable a	SEPL Expression	Equivalent To	Returns
<a xmlns:ns="...">	a.b.c[0]	a.xpath("b/c[1]/text()")	"text1"
<ns:b num="1">	a.b.c[1].attr("id")	a.xpath("b/c[2]/@id")	"abc"
<c num="2">c1</c>	a.b.c	a.xpath("b/c")	{"c1", "c2"}
<c id="abc">c2</c>	a.b	a.xpath("b")	XML element
</ns:b>	a.b.attr("num") +	a.xpath("b/@num") +	3
	a.b.c[0].attr("num")	a.xpath("b/c[1]/@num")	

SEPL supports asynchronous service invocations with WS-BaseNotification [25]. The client subscribes at the service to receive notifications with a certain content (specified using XPath). SEPL provides a convenient way to handle notification registrations and events. In Figure 3, the UML *CallBehaviorAction* named `subscribe` registers a notification subscription and appends the returned callback object to the array `callbacks`. Within this subscription, the callback object will receive all notifications matching the XPath given as the first argument input pin. The XPath points to an XML element with name `PortingResult` and an attribute `customer` with the respective customer identifier. The string `$1` is a placeholder for the first argument after the XPath (`r.customer`). Further arguments are referenced using `$2`, `$3` etc. After subscribing for a notification, execution continues until a *CallBehaviorAction* named `wait` is activated, which takes the callback as an input. This action blocks until the service sends a matching notification. An optional timeout parameter can be specified to prevent the `wait` action from blocking endlessly if no notification arrives. The predefined `async` object provides access to additional configurations such as the listening port (not used in the sample).

SOAP Faults, the Web service equivalent to exceptions in ordinary programming languages, are messages with a well-defined syntax which are sent by services to indicate that an error occurred while processing a request. A SOAP fault message contains a fault code, and both a brief and a detailed description of the

fault's reason and its origin. SEPL provides means to handle SOAP Faults using a UML *ExceptionHandler*. The handler is depicted as an arrow (resembling a lightning bolt) and points from the action in which the fault may occur to the fault handling action. Fault handling always happens for a particular fault code (`invalidCredentials` in the sample), although it is possible to use a wildcard symbol (`*`) to catch faults of any type. Inside the fault handling action the predefined object `fault` can be used to obtain details about the fault. SOAP Faults can also be thrown from inside the protocol function using the `throw` action.

3 Generating Executable SEPL Code

The model representation of SEPL protocols in the form of UML activity diagrams provides the required information to generate executable SEPL scripts. In the following we present the syntax of this domain-specific script language on the basis of the scenario protocol. We then define the mapping between UML elements and SEPL script code. Finally, we briefly discuss the code generation algorithm.

3.1 Scenario Service Protocol

Listing 1 prints the executable SEPL code representation of the sample scenario. The script is generated from the scenario UML model in Figure 3. In this section we only briefly discuss the code syntax. The complete syntax rules for SEPL code in EBNF (Extended Backus-Naur Form) are listed in the Appendix (Section 9). Line numbers in the following subsections always refer back to Listing 1.

```
1  factory.wsdl = "http://infosys.tuwien.ac.at/Factory?wsdl"
2
3  function port_numbers(username, password, requests) {
4   this.EPR = factory.createResource()
5   try {
6     login(username, password)
7   } catch (invalidCredentials) {
8     result = fault.detail
9     Destroy()
10    return result
11  }
12  for (r : requests) {
13    status = check_porting_status(r.customer,r.newProvider)
14    if(!status.isPossible) {
15      rollback_all()
16      throw Fault(status.details)
17    }
18    properties.customer = r.customer
19    properties.newProvider = r.newProvider
20    callbacks[] = async.subscribe(
21       "//PortingResult[@customer='"+r.customer+"']")
22    schedule_porting_for(r.time)
23  }
24  for(c : callbacks)
25    result[] = async.wait(c)
26  Destroy()
27  return result
28 }
```

Listing 1 Number Porting Service Protocol in SEPL Code

In line 1 of the listing, the WSDL location of the factory service is set. Note that the WSDL location is not contained in the UML representation in Figure 3 but provided as an additional parameter when the SEPL code gets generated. Line 3 marks the start of the code of the protocol function `port_numbers`. First, the factory method is invoked and the resulting EPR is stored with the predefined object `this` (line 4). Lines 5-11 contain the login invocation, along with a fault handling routine in a `try` and `catch` notation similar to the exception handling syntax in traditional programming languages. Lines 12-23 embrace the `for` loop which handle each item in the array `requests`. If the status result obtained from the invocation in line 13 is not affirmative (line 14), the complete number porting operation is rolled back using the invocation of operation `rollback_all` (line 15) and a SEPL protocol execution fault is thrown (line 16). Lines 18-19 set the two WSRF resource properties `customer` and `newProvider`. The notification subscriptions are added with lines 20-21 by means of the predefined object `async`. Note that in the XPath expression the string `$1` occurring in the UML representation has been replaced with the according parameter (`r.customer`). After all portings have been scheduled for asynchronous execution (line 22), the code loops over all callbacks to wait for the result (using the `async` object, lines 24-25) returned from the service using WS-BaseNotification. Finally, the created resource is destroyed (line 26) and the result variable gets returned (line 27). Note that these two lines of code are generated twice (9-10, 26-27), because in the UML representation the `Destroy` invocation is accessible both from the second loop node and the fault handler `assign` action.

3.2 UML-to-SEPL Mapping

Table 3 gives an overview of the mapping between SEPL language constructs and PAD elements. Column 1 contains the name or purpose of the language construct, column 2 gives the name of the corresponding UML element, and column 3 shows the SEPL code representation based on an example. Reserved variable names and keywords are in bold print.

The main entity of SEPL PADs is the UML activity, which is mapped to a `function` in SEPL code. Just as a PAD may contain several activities, a SEPL script may contain several functions. Service invocations in UML are modeled with *CallOperationAction* elements that are associated with input pins (operation parameters) and the optional output pin `result` to assign the result of an invocation to a variable. In SEPL code, invocations resemble a method call in an ordinary programming language. As has been shown in Figure 4, upon execution of a protocol service invocation a SOAP message is constructed and sent to the target service; the response is transformed back to the SEPL code representation and can be directly assigned to a variable. The return value of a protocol activity in UML is indicated with an output *ActivityParameterNode*. In the generated code, a `return` statement is inserted at the end of the protocol function. Access to resource properties both in UML and in SEPL code is provided by the predefined object `properties`. The assignment construct follows the standard notation of a *AddVariableValueAction* in UML, and is mapped to an assignment expression using an equality sign (`=`) in SEPL code. The base case is that a variable gets overwritten, in which case the UML attribute `isReplaceAll` is set to `true`. Otherwise,

if this attribute evaluates to `false`, the corresponding SEPL code uses a special syntax to append a value to an array, e.g., `result[]=...` .

Table 3 SEPL-to-UML Mapping

Language Construct	**UML Element**	**SEPL Code (Example)**
Protocol Function	*Activity*	**function** f1(param1,param2){ ... }
Service Invocation	*CallOperationAction*	login(username,password)
Invocation/Assignment	*CallOperationAction* with *result*	result = login(username,password)
Return Value	*ActivityParameterNode*	**return** result
Simple Assignment	*AddVariableValueAction*	i = i + 1
Set Resource Property	*AddVariableValueAction*	**properties**.prop1 = value
Subscribe Notification	*CallBehaviorAction* 'subscribe'	callback=**async.subscribe**("//result")
Receive Notification	*CallBehaviorAction* 'wait'	result = **async.wait**(callback)
If-Branch	*DecisionNode* [& *MergeNode*]	**if**(var1 < 10) { ... }
If-Else If-Else-Branch	*DecisionNode* [& *MergeNode*]	**if**(...){ ... }**else if**(...){ ... }**else**{ ... }
Array Iteration	*LoopNode* with *loopVariable*	**for**(r : requests) { ... }
Loop	*LoopNode* with *decider*	**while**(var1 < 10) { ... }
SOAP Fault Handling	*ExceptionHandler*	**try**{...}**catch**(invalidCredentials){...}
Protocol Fault	*CallBehaviorAction* 'throw'	**throw Fault**(status.details)

Protocol faults and the actions to subscribe for and receive notifications are modeled as UML *CallBehaviorAction* elements. Depending on the name of the element (`throw`,`subscribe`,`wait`) the corresponding SEPL code is produced (see Table 3). Decision branches are modeled in UML using a *DecisionNode*, where each branch holds a *guard expression* denoting the associated condition. A *MergeNode* may be optionally used to merge two or more branches and to continue with a single control flow. The corresponding SEPL code contains `if` and `else if` blocks with the conditions of the guard expressions, and optionally an `else` block for a branch without guard. UML *LoopNode*s are used to model both loops and array iterations. Loops hold a *decider* output pin, which gets evaluated before each iteration. To iterate over all elements of an array, the *LoopNode* holds a *loopVariableInput* input pin combined with a *loopVariable* output pin. Loops are represented as `while` blocks and the code syntax for array iteration is a `for` block as indicated in Table 3. Finally, SOAP fault handling in PADs is modeled using an *ExceptionHandler* edge, which points from the protected action node to a handler body node. The protected code is put in a `try` block and the instructions of the handler body node are embraced by a `catch` block, which specifies the Fault code to be handled (`invalidCredentials` in the sample) or a wildcard symbol (`*`) if any fault should be caught.

3.3 UML2SEPL Code Generator

In order to create SEPL source code from a PAD (service protocol activity diagram), we implemented the *SEPL code generator* (*UML2SEPL*), which is briefly discussed in the following. The *Eclipse Model Development Tools* (MDT) [10] serve

as the platform for SEPL modeling. With MDT, UML models can be saved in an XML format based on *XML Metadata Interchange* [24] (XMI). The code generator consists of two main parts: 1) a *UML factory* which reads encoded UML files and builds an in-memory object representation (see Figure 6) and 2) the actual *code writer*, which outputs the relevant SEPL code to an output stream. By making this distinction we strive for independence from the UML notation (the file format in which the PAD is saved). New notations can be integrated by implementing an

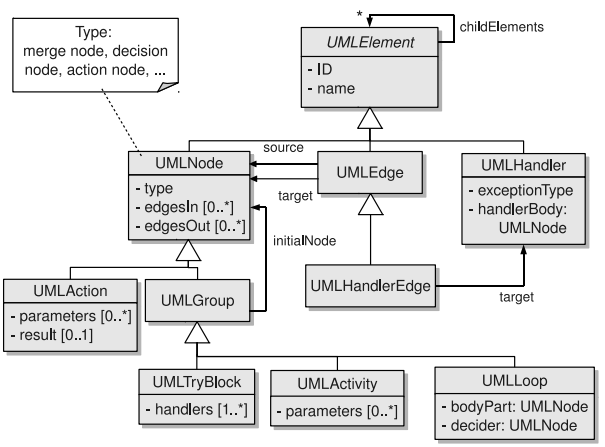

Fig. 6 Class Diagram of UML PAD In-Memory Representation

appropriate factory class. Our implementation is based on the file format used by the *Eclipse Modeling Tools* [10] (MDT). With MDT, users can graphically design SEPL PADs in the Eclipse IDE and save the model to a file.

We will not discuss in detail how the in-memory model is created from the MDT AD file as this involves mainly parsing the XML markup. A complete discussion of the code writer algorithm is also out of the scope of this paper. In short, the proceeding of the code writer is similar to the visitor pattern [15]: basically it iterates over all activities and calls a method `visit`, which recursively visits all nodes and outputs corresponding SEPL code.

4 Service Protocol Hosting

The SEPL client implementation, which is discussed in more detail in Section 5.1, allows for the client-side execution of service protocols. Client-side protocol execution has some drawbacks. For one thing, asynchronous communication with services requires the client to open a separate port to listen for notification messages. Secondly, client applications require the SEPL client library files, in addition to standard Web service libraries. We therefore provide a solution for clients to launch a protocol execution with a standard SOAP client. This is accomplished by setting up a server on which the service protocols are published as Web services themselves. We refer to this part of the framework as *SEPL Protocol Host* (PH).

Figure 7 depicts the PH providing the number porting protocol introduced in Section 1.2. The number porting Web service is deployed on the service provider side, and a SOAP client is available on the client side. With the combined information of the WSDL and SEPL documents the PH generates and publishes the *Porting Protocol Web Service*, which provides the protocol function `port_numbers`. The description of the protocol service is published in the *Porting Protocol WSDL* document, which contains the functions of the SEPL document as WSDL operations. The client uses a standard SOAP implementation to parse the service protocol WSDL and send an invocation message to the Porting Protocol Web service. This service receives and unmarshals the SOAP message and delegates the request to the PH for execution. Note that the PH is typically published by the service provider, although it is possible that the PH remains with a third party (hence the dashed line between PH and the service).

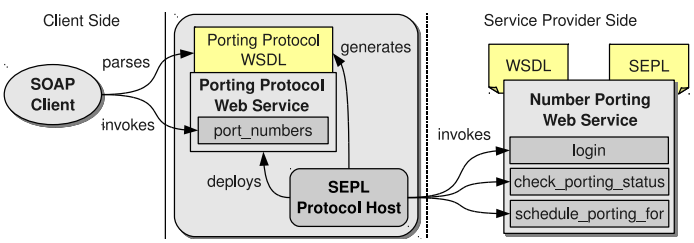

Fig. 7 SEPL Protocol Host

We identify three main tasks performed by the PH:

1) generating the service protocol WSDL document from the service's SEPL and WSDL documents;
2) dispatching incoming requests;
3) executing the protocol and returning the result.

The issue with point 1) is that SEPL function parameters are untyped and that WSDL, being based on XML, requires XSD type information of operation parameters. Therefore, we developed an algorithm that attempts to determine the data type of parameters and return values of SEPL functions (see Section 5.2 for more details). Using this type information, the WSDL generator outputs a WSDL operation for each SEPL function, as well as according XML schema definitions. Point 2) is achieved by using a WS-Addressing `Action` element of the format `<protocol name>:<function name>`, which helps to uniquely identify a service protocol function. This `Action` element is included in the `portType` element of the generated protocol WSDL contract. WS-Addressing enabled clients will parse the WSDL and automatically include the appropriate `Action` header in their SOAP messages. For point 3), the actual execution of the protocol, we make use of the SEPL execution client, which will be discussed in more detail in Section 5.1.

Figure 8 illustrates the service protocol WSDL generation based on the implementation of the number porting scenario (compare Section 3). The binding style of the `Number Porting` service's WSDL is *document/literal wrapped* [6], i.e., the operation parameter elements in the XSD are "wrapped" in elements having the same name as the operations they are part of. The details of the `message`,

`portType` and `binding` sections are omitted for brevity. For the WSDL generation, with regards to the example depicted in Figure 8, we consider the following:

- The name of the XSD top element (wrapper element) equals the name of the protocol function (`port_numbers`).
- The parameters `user` and `pass` are passed to the invocation of the operation `login`. The XSD types of this operation's parameters are both `string` (see the WSDLs `types` section), hence the type of the function's parameters `user` and `pass` are assumed to be of type `string`.
- The parameter `requests` is used in a `for` loop, which indicates that it is an array (or sequence) of elements. In the generated WSDL this is specified using the attributes `minOccurs` and `maxOccurs`.
- The function `port_numbers` returns an array of elements which are received using notifications at run time. Since the type and content of the notification messages is not known at design time, the type of the return message in the generated WSDL document is `any`.

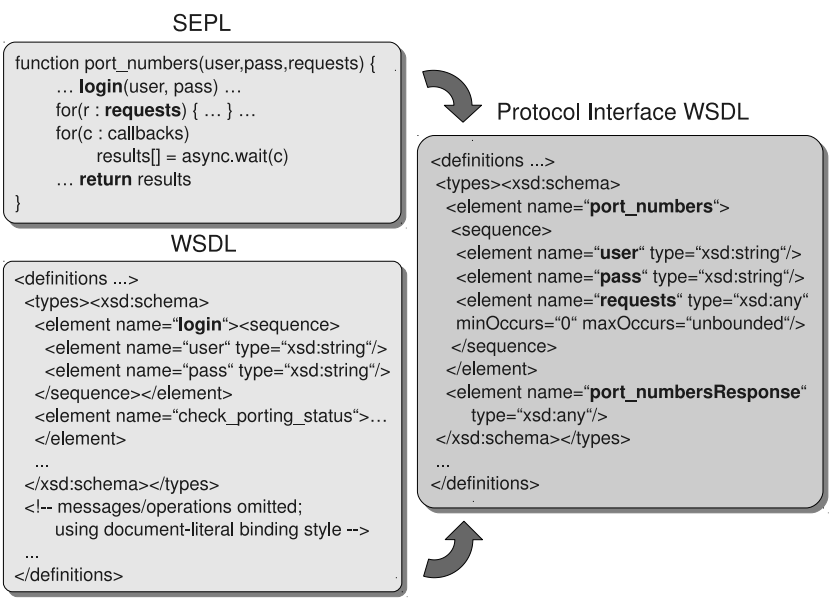

Fig. 8 Service Protocol WSDL Generation

5 Framework Implementation

In this section we discuss the implementation of the SEPL framework. Firstly, we take a look at the "big picture", i.e., how the framework components are connected with each other, and the details will be explained in the subsections to follow. Figure 9 illustrates the implementation of the example scenario presented in Section 1.2. The scenario is based on the functionality of the `Number Porting` Web service

which supports porting of mobile numbers across providers. The static interface of this service is defined in the WSDL contract, the functionality is laid down in an according SEPL document. The *SEPL Protocol Host* (PH) is responsible to execute the SEPL protocol and to expose its functionality to a *SOAP Client* (which represents the end-user) in the form of invokable WSDL operations. The PH is implemented as a Java Web Application [35] and is deployed in a Tomcat application server [1]. The *WSDL Generator* is responsible to generate a WSDL definition containing the Web service interface of the protocol functionality. The *UML2SEPL Code Generator* converts SEPL activity diagram (PAD) models into SEPL code – an optional preprocessing step which is necessary in case the `Number Porting` service is configured with a PAD model instead of a SEPL source document.

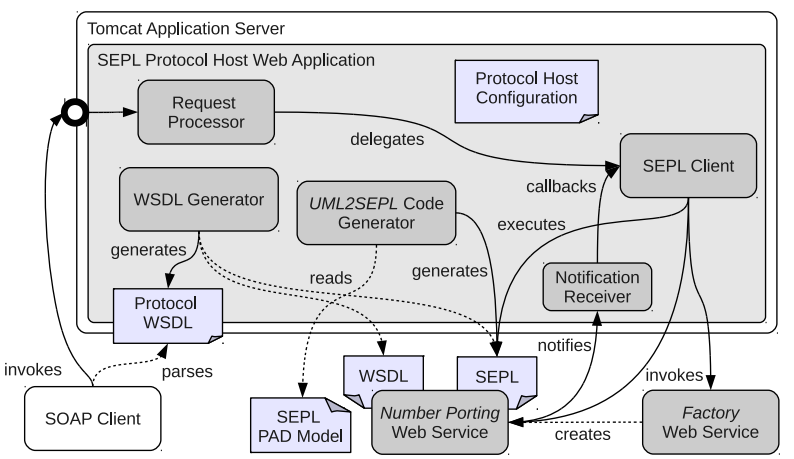

Fig. 9 Connection Between the SEPL Framework Components

If the end-user decides to execute a protocol function, a regular SOAP client is used to parse the *Protocol WSDL* and to send an according SOAP message to the PH. The *Request Processor* receives the SOAP message and dispatches it. Then the request is delegated to the *SEPL Client* which is embedded in the Web application. The SEPL client reads the SEPL document and executes the protocol by invoking operations of the `Factory` and the `Number Porting` Web services. When the execution reaches a `wait` statement, the SEPL client waits until the service sends a notification message, which is received by the *Notification Receiver* and handed to the SEPL client. The result of the protocol execution is handed back to the request processor, which returns it to the SOAP client.

5.1 SEPL Execution Client

The SEPL client implementation has been developed in the Java programming language. Figure 10 illustrates the structure of the engine and the responsibility of

[1] http://tomcat.apache.org

the separate parts, in the context of the example `Number Porting` Web service and the service factory. In preparation of SEPL documents for interpretation, the *Code Preprocessor* reads SEPL protocol files and applies the necessary modifications to convert the DSL constructs into the format of a concrete *host language* (i.e., SEPL is a hybrid between *embedded* and *external* DSL [34]). The modified source code is interpreted by a *Code Interpreter*, which directs the control flow of the protocol and maintains the state of the variables. Currently, *Pnuts* [2] is used as the host language and the interpreter is implemented as an extension of the *Pnuts* scripting engine. All Web service specific tasks - as part of the *Core Execution* - are delegated by the code interpreter to the respective specialized parts of the client. The *WSDL Parser* reads WSDL documents and parses them for operations, their parameters and for WSRF resource property definitions. The *SOAP Stack* performs all Web service invocations and mediates between the SOAP messages on network level and Java objects at high level. The SOAP stack is implemented by *Daios*, an efficient framework for dynamic Web service invocation, which has been measured against other popular Web service clients with good results [18]. In the figure, arrows point from the execution engine to the target Web service and the factory Web service to illustrate the direction of invocation. The arrow in the opposite direction signifies the flow of WS-BaseNotification messages which are received and processed by the *WSN Processor*.

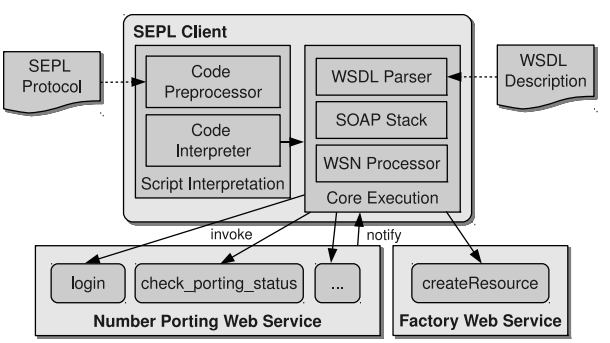

Fig. 10 SEPL Client Architecture

The Code Interpreter is based on the Java scripting engine Pnuts, which is extensible in so far that it supports the definition of new object types with classes that implement the interface `pnuts.lang.AbstractData`. These custom classes can be seen as the DSL-specific extension to the Pnuts core. Pnuts is responsible to parse and interpret SEPL code, whereas the SEPL extension performs domain-specific tasks such as WSDL parsing, Web service invocations, XML processing and so forth. In order for Pnuts to interpret SEPL code, the source needs to be adapted by the Code Preprocessor. Pnuts cannot handle, for instance, faults in the way they are syntactically defined in SEPL `catch`-blocks. As a solution, the preprocessor dynamically defines exception classes which represent the SOAP fault codes. Moreover, all string occurences are wrapped with a call to an initialization method, which creates an `XML` instance in case a string contains valid XML markup.

[2] https://pnuts.dev.java.net

5.2 SEPL Protocol Host

The SEPL Protocol Host (PH) is a Java Web Application based on the Web Services Engine *Axis2*[3]. But, unlike in Axis2 where services are statically configured, SEPL configures all required Web services dynamically on deployment. A file `sepl.xml` contains the PH configuration, which, among other things, defines the host and port on which the PH runs, the endpoint of the notification service to be used and a list of service protocols which are handled by this PH. Each service protocol is described as a tuple $(name, wsdl, sepl)$ containing its unique name, the URL of the target service's WSDL file and the location of the SEPL specification (either in source code or as a UML model).

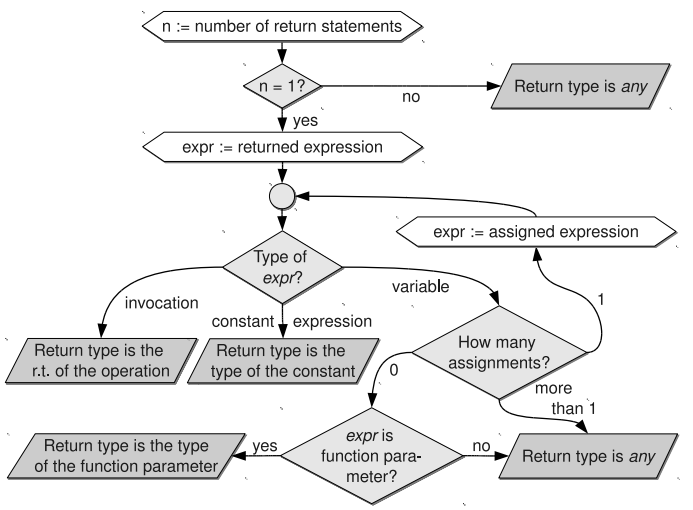

Fig. 11 Determining the Return Type of a Function

Upon initialization of the PH Web application, the WSDL generator is used to construct the WSDL documents of all configured protocols. As has been mentioned in Section 4, a specialized algorithm tries to determine the data types of parameters and return values by means of a static analysis of the SEPL code. Figure 11 depicts a flowchart containing the programmatic decisions made to determine the return type of a SEPL function from the source code. Firstly, the source code is checked with a regular expression to find out the number of return statements contained therein. If more than or less than one return statement exists, the return type is set to `any` (unknown). Otherwise, the (single) `return` statement is syntactically analyzed. If the statement returns the result of an operation, the functions return type is the return type of the operation (which can be extracted from the WSDL). If the statement returns a constant value (a string, numeric value or boolean value) then the return type is the type of the constant. If the statement returns a variable, we have to make another case distinction. If the variable gets assigned exactly once in the function (and assumingly *before* the return statement), a recursive call is

[3] http://ws.apache.org/axis2/

made and the assigned variable is further analyzed. If the variable gets assigned more than one time, we set the return type to `any`. If no assignment to the variable in question is found in the function, the variable must be a function parameter, in which case the return type of the function equals the type of the respective parameter.

Determining the types of function parameters is less complicated than the types of return values. We mentioned in Section 2 that parameters are final, thus an assignment to a parameter variable is invalid and the type of parameters is preserved throughout the function. It is therefore sufficient to find one Web service invocation instruction in which the function parameter is used as a parameter to the invocation. This is implemented in the PH with the aid of Java regular expressions. If a parameter type cannot be determined by this means, its XSD type in the WSDL is set to `any`.

Once the protocol WSDL has been generated and published, clients use it to construct SOAP invocations to the PH. The *Request Processor* component intercepts all invocations, reads the WS-Addressing `Action` header (which contains the protocol name) from the incoming message and delegates the request to a SEPL client instance for execution.

6 Evaluation

In this section we perform an evaluation of the work presented in this paper. The evaluation targets two aspects: one part is concerned with qualitative and quantitative characteristics of the approach in general. In this part we spotlight the size and readability of SEPL service protocols, as well as efficiency criteria (e.g., speed of development, ease of maintenance). The second part analyzes the SEPL framework performance based on an end-to-end scenario implementation, and compares the performance to other possible solutions.

6.1 Qualitative and Quantitative Characteristics of SEPL

In the following we compare the characteristics of the SEPL language and execution framework to other possible solutions for specifying and executing intra-service protocols.

Consider the example number porting service protocol presented in Section 1.2. In principal, this scenario protocol could be specified in various ways. Our evaluation compares the following possible techniques to solve the scenario, which target the same goal but nonetheless constitute very diverse methods:

- hardcoding the solution, i.e., creating a client which implements the business logic, or extending the service itself by adding a new operation. Note that a hardwired solution may be generally undesired since it contradicts the SOA paradigm of loose coupling [29].
- defining the protocol using the Web Services Conversation Language [36] (WSCL). WSCL extends the static description of Web services (WSDL) and allows to define conversational aspects such as the order in which messages need to be exchanged. WSCL is not an executable protocol, but a guideline for the interaction with a service (for details see related work in Section 7).

- creating a WS-BPEL process, which implements the functionality. The process is decoupled from the target service and deployed in a WS-BPEL engine, which is responsible for execution the protocol functionalities.
- developing and publishing the protocol in SEPL as described in the course of this paper.

An overview of the comparison is printed in Table 4. An important distinction is that protocols expressed in WSCL are not directly executable, whereas the other variants provide all necessary details for execution. The fact that WS-BPEL and WSCL are accepted industry standards indicates that a lot of experts are trained in these languages. However, for a person with some (Web) programming background SEPL can be learned with reasonably low effort.

Concerning the syntax, SEPL comes in two flavors: graphical UML representation and script code. WSCL and WS-BPEL use an XML-based syntax, and the custom client implementation may be developed in a general purpose programming language (GPL) such as Java. SEPL code is more light-weight and requires less instructions to implement a protocol compared to the alternative solutions. For the presented number porting functionality, SEPL requires only around 20 language constructs (compare Table 3), as opposed to the hardcoded client-side implementation (using the Daios framework) with more than 70 lines of code (LOC), the WS-BPEL process with roughly 50 instructions and the WSCL rules with roughly 40 language constructs. The XML-based syntax of WS-BPEL is harder to read for humans, but XML is well suited to be processed by machines. Graphical development tools exist for WS-BPEL and SEPL (editor for UML activity diagrams), but are generally not available for the other variants.

SEPL has built-in support for WSRF (resource creation/destruction, identification using WS-Addressing, access to resource properties). WSCL does not take WSRF into account, and also WS-BPEL has no direct support for WSRF, which has been addressed in previous works [12, 8].

Table 4 Comparison of Service Protocol Implementation Variants

	Client Impl. / Extend WS	WSCL	WS-BPEL	SEPL
Syntax	GPL	XML-based	XML-based	UML / script
Executable	✓	×	✓	✓
Graphical Tools	×	×	✓	✓
WSRF Support	client library	×	×	✓
Standardization	×	✓	✓	×
Modularization	possible	×	×	✓ (`functions`)
Development Speed	slow	medium	slow	fast
Maintenance	recompile	unconstrained	redeploy	ad-hoc
Dyn. Correlation	tailor-made	×	×	✓
Loose Coupling	×	✓	✓	✓
Impl. Size (ex.)	$>$ 70 LOC	$\sim$ 40 constructs	$\sim$ 50 constructs	$\sim$ 20 constructs

Modularization is easy with SEPL, since SEPL protocols can be split up into `functions`, which can be invoked from one another. WS-BPEL does not explicitly allow for the use of subprocesses. However, since they are also exposed as Web services, WS-BPEL processes are recursively composable. As far as maintenance in concerned, developers face different degrees of flexibility when a modification in an existing service protocol function becomes necessary. Custom client or Web service implementations usually require recompilation of the affected code parts, and a modified WS-BPEL process needs to be redeployed in the execution engine. As WSCL is not executable but merely descriptive, no runtime constraints need to be considered. Since the script code representation of SEPL protocols is directly interpreted, changes can be incorporated in an ad-hoc fashion at runtime. Our experience with development of service protocols has shown that SEPL brings an enhancement in development speed and productivity, a characteristic that is in general often attributed to DSLs [34]. Custom implementations of service protocols or WS-BPEL implementations contain many subtle technological challenges, e.g., related to XML, XPath or Web service technologies. SEPL abstracts from these implementation details and provides a more readable, easy to use DSL.

Concerning asynchronous invocations, WS-BPEL imposes a difficulty: in order for the execution engine to correlate an incoming notification message with a certain process instance, the process definition needs to define a `correlationSet` with correlation properties. However, the number of correlation property instances needs to be defined at design time, which complicates the implementation of dynamic correlations. Such correlations for asynchronous invocations are provided by the SEPL callback mechanism.

Table 5 SEPL versus WS-BPEL Language Constructs

SEPL	**WS-BPEL**	**SEPL**	**WS-BPEL**
Service Invocation	`<invoke .../>`	Protocol Function	(`<process>...`)
Return Value	`<reply .../>`	Set Resource Property	(`<invoke .../>`)
Sequential Control Flow	`<sequence>...`	Subscribe Notification	(`<invoke .../>`)
Simple Assignment	`<assign><copy>...`	Receive Notification	(`<receive .../><if>...`)
If-Else If-Else-Branch	`<if>...`	(Receive Notification)	`<pick>...`
Array Iteration	`<forEach ...>...`	n/a	`<partnerLinks>...`
Loop	`<while ...>...`	n/a	`<flow>...`
SOAP Fault Handling	`<faultHandlers>...`	n/a	`<correlations>...`
Protocol Fault	`<throw ...>...`	n/a	`<wait>...`
Scope (*StructuredActivity*)	`<scope>...`	n/a	`<validate ...>...`

As a final aspect of the qualitative evaluation we compare the language features of SEPL and WS-BPEL in Table 5. The left part of the table lists features that are available in a similar fashion in both languages, e.g., the service invocation in SEPL and the WS-BPEL `invoke` element. The right part of the table lists the language differences: features that are only partly available or can be achieved using a workaround are printed in parantheses, and features that are not available are indicated with `n/a`. SEPL protocols can be split up into functions, whereas BPEL does not support explicit modularization. Manipulating WSRF resource properties has

to be performed manually in WS-BPEL by invoking the according getter/setter operations. Similarly, notification subscriptions are achieved using custom invocations. To receive a notification with a certain message content, WS-BPEL requires a rather cumbersome combination of `receive` and `if` elements. Additionally, message correlation has to be manually defined, which is often prone to errors. The WS-BPEL `pick` activity is used to wait for one of several possible messages, which in SEPL can only be solved using a workaround with notifications. Finally, a number of WS-BPEL language constructs are not available in SEPL. Selecting from different service endpoints (`partnerLinks`) is not required in intra-service protocols, parallel execution (`flow`) is not beneficial. Message `correlations` are a feature for long-running business process conversations and deliberately not supported in the light-weight SEPL language. Also the WS-BPEL activities `wait` and `validate` are not provided in SEPL.

6.2 Framework Performance

In the following we evaluate the SEPL framework based on the performance tests that we carried out. The aim of the tests is to determine how much overhead the (dynamic) interpretation of SEPL code causes, in comparison to (static) implementation of the protocol directly using a SOAP client (Daios) and in comparison to an implementation using WS-BPEL. We consider the example protocol presented in Section 1.2 in a slightly modified version, leaving out the asynchronous messaging part (subscription and notification of finished portings). The tests are executed in seven levels with an increasing number N of number porting requests ($N \in \{10, 50, 100, 200, 400, 700, 1000\}$). All tests have been run ten times and the numbers presented in the following are average values. The test execution has been performed on a computer with a quad core 2.8GHz processor and 4GB RAM, under the Linux operating system Ubuntu 9.10 [4] (Linux kernel 2.6.31-17). The WS-BPEL implementation of the process has been deployed in a Sun Glassfish [5] application server (version 2.1.1) using the `sun-bpel-engine` module.

Table 6 Performance Test Results

	Daios		SEPL		SEPL PH		↔	WS-BPEL	
#	Avg.	S.D.	Avg.	S.D.	Avg.	S.D.	t	Avg.	S.D.
10	52.1	7.6	327.6	14.9	874.8	158.8	**1.8**	1140.7	418.3
50	277.6	79.1	588.5	88.1	1044.6	156.4	**2.2**	1388.6	439.0
100	482.7	98.3	829.0	117.3	1288.0	164.8	**2.6**	1692.6	436.2
200	873.8	34.8	1295.9	54.2	1781.3	171.0	**3.2**	2298.0	452.7
400	1761.9	48.8	2370.3	77.5	2853.9	134.9	**4.6**	3526.0	421.9
700	3039.2	68.4	3903.2	92.5	4412.3	153.2	**5.4**	5463.6	562.3
1000	4501.4	461.1	5515.7	127.0	6000.5	315.4	**9.6**	7712.9	433.3

[4] http://www.ubuntu.com
[5] https://glassfish.dev.java.net/

The results of the test runs (average run times and standard deviations) are listed in Table 6. `Daios` denotes the time consumed by the Web service invocation framework. The difference between the values for `SEPL` and `Daios` reflects the overhead of the SEPL client, including SEPL code preprocessing, data transformation and the actual SEPL code interpretation. The figures for `SEPL PH` are slightly above those for `SEPL`, a difference that results mainly from the network transfer as well as (de-)serializing of the messages exchanged between the client and the SEPL PH.

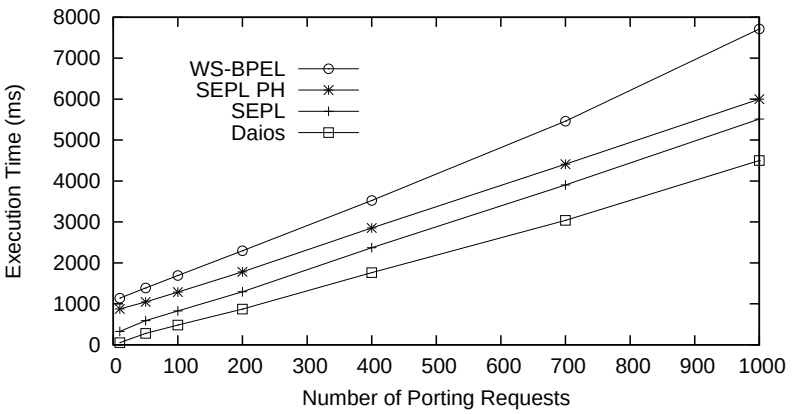

Fig. 12 Execution Time Trendlines for Scenario Protocol

To check the statistical significance of the results, most importantly of the difference between `SEPL PH` and `WS-BPEL`, we performed a *t-test* for two independent samples [19]. The t-test assumes that the test data is normally distributed. Hence, we conducted a Anderson-Darling goodness-of-fit test [7], which revealed that, with a `95%` confidence interval, it is unlikely that the data is not normally distributed. To compare implementation versions A and B, we calculate the value `t`, which is defined as $t := \frac{\bar{x}_A - \bar{x}_B}{\sqrt{\frac{(s_A)^2+(s_B)^2}{n}}}$, where $\bar{x}_A$ and $\bar{x}_B$ are the average (mean) run times of A and B, s_A and s_B are the estimated standard deviations of the run times of A and B, respectively, and n is the number of iterations (`10`). The value of `t` is compared to the critical value of `t`, which is obtained from the *t-distribution*. We target a `95%` confidence interval and the *degree of freedom* is $(n-1)+(n-1) = 18$. Hence, the critical value of `t` is `t(0.95, 18) = 1.734`. As can be seen in Table 6, the calculated value of `t` is greater than the critical value (`1.734`) in all test levels, hence the difference in runtime of `SEPL PH` versus `WS-BPEL` is significant in this test scenario. The value of `t` increases with rising test level, which suggests that the SEPL PH scales at least as good as or even better than the WS-BPEL engine for large requests. It should be noted that, despite the fact that SEPL performs best in our evaluation scenario, we do not claim that SEPL is generally superior for all types of protocols and all WS-BPEL implementations.

7 Related Work

The problem of service protocol definition as discussed in this paper is essentially a subset of the service composition problem. Modeling and description of both service protocols and compositions can be achieved in various ways. For example, [33] models business protocols using finite state machines. To that end, a business workflow is divided into a set of states with one initial state and a set of final states. A transition relation defines which states are accessible from a source state and which message is consumed when a new state is reached. Also the service protocols in [3] are modeled as state machines. Based on a formal model to express protocols, the paper focuses on commonalities and differences between protocols (protocol management operators) and compatibility issues when combining or replacing different protocol definitions. In contrast to this theoretical and foundational work in terms of its general applicability, the focus of SEPL is to define directly executable protocols tailored to Web services and XML data. State machines are not well suited for our purpose because SEPL protocols need to exactly define the input-output transformations of data passed from one operation to another. For modeling service protocols SEPL therefore relies on UML activity diagrams which provide both the desired abstraction and level of granularity.

The approach in [17] uses a Petri net based algebra as a theoretical framework for composition of Web services. Each place represents a state, the transitions are the operations of a service and directed arcs define the flow relation between two states. A similar Petri net based approach has been introduced in [37]. The advantage of Petri nets lies in the support for many flow concepts (e.g., choice, parallelism, iteration) and its formal foundation. Well-known algorithms can be applied to prove the correctness (e.g., termination, reachability of places) even for very complex compositions. On the other hand, Petri nets are unhandy to use and can grow unmanageably large even for small or mid-size scenarios. Similar problems arise when using formal process models based on the *pi*-calculus [32] for composition or protocol specification.

The most prominent, de-facto standard service composition language is WS-BPEL [26], which offers a broad spectrum of operations using an XML-based syntax. WS-BPEL has a broad vendor support and is popular for its applicability to most composition and protocol definition scenarios. Unfortunately, WS-BPEL falls short of supporting stateful service resources, especially their creation using factories, in an appropriate way. [12,8] state that WS-BPEL does not define a standard way to store the resource identifier returned by a factory service and automatically use it in subsequent invocations on the created resource. In [8] a WSRF-specific `gridInvoke` operation is suggested to overcome this issue.

Other related works have been published in the field of service mashups [4], which create new functionalities by combining services and data from heterogeneous Web resources. For instance, the IBM Sharable Code platform presented in [20] provides a structured DSL for defining, sharing and executing Web service mashups. The rise of service mashups constitutes a trend towards an open programmable Web. The basis for mashup development is an exact description of the static and dynamic service interface, for which service protocols play an important role. Whereas mashups are tailor-made applications and usually created in a community-driven fashion, SEPL service protocols are an integral part of the service description that can be (re-)used by mashup platforms and clients directly.

The authors of [13] present *XL*, an XML programming language for Web service specification and composition. It is argued that current Web service implementations have integration deficiencies: host programming languages such as Java or Visual Basic in combination with XML documents and back-end (relational) database management systems build up a heterogeneous environment with difficulties. XML data must be converted to Java objects and vice versa. Java objects must be marshaled through database management interfaces (e.g., JDBC). XL attempts to address these issues and provides features to specify Web service implementations. Similar to SEPL functions, XL defines *operations* which describe service functionalities using control flow directives (`if, switch, while, for`), invocations of (external) service operations and input-output transformation. XL and SEPL are similar in the way they handle XML data as both languages directly integrate XML processing in the syntax. Both languages support *XPath* to access certain elements and attributes of XML markup. SEPL additionally supports a "dot-syntax" which resembles the syntax to access class members in object-oriented programming. XL supports *XQuery* statements, which operate on XML data sources and serve as a replacement for queries to external databases. In general, XL is designed to contain much of the business logic and does not necessarily require an existing target service whereas SEPL documents are rather slender and delegate most tasks to the target service. XL and SEPL use different conversation patterns: XL requires a *conversation-URI* header in each exchanged SOAP message to identify which conversation the message belongs to, which imposes requirements on the clients' capabilities; SEPL, on the other hand, creates new service instances where needed and publishes the functions as stateless operations which do not require clients to consider any conversation-specific aspects. XL allows for parallel execution which is not supported in SEPL. In SEPL it is possible to perform asynchronous invocations, which XL does not directly support.

The *Web Services Conversation Language* (WSCL) [36] is an effort to extend the standard Web service description (WSDL) by conversational aspects. The WSCL specification declares that *"defining which XML documents are expected by a Web service or are sent back as a response is not enough"*. Beyond the mere description of the input and output messages, WSCL defines the order in which they may be exchanged. WSCL is helpful to model intra-service dependencies in a general way and to lay down the order in which interactions may occur, but fails to specify how the interactions are connected, i.e., how the result of one interaction can become part of the input to the next interaction. In SEPL this is possible – input and output can be transformed directly and arbitrarily. Furthermore, WSCL allows only for distinctions concerning the type and not the actual content of messages. In total, WSCL does not define executable protocol functionalities but is rather a guideline for the interaction with a service.

In WSDL 2.0 the concept of Message Exchange Patterns (MEPs) has been introduced. MEPs can be seen as simple general-purpose intra-service protocols and are therefore related to our work. However, the MEPs predefined in WSDL 2.0 are rather simplistic (in-out, in-only, robust-in-only, ...). Additionally, in WSDL 2.0 MEPs are described in an informal human-readable format, and are not suitable for machine interpretation. The concept of MEPs has been extended in SSDL [30], where arbitrarily complex MEPs are used to define protocols and contracts between services. However, SSDL has a rather different focus than the work we present in this paper – SSDL has been proposed as an alternative to service com-

position languages such as WS-BPEL, and seems not well-equipped to specify the intra-service message exchange of single services as discussed in this paper.

8 Conclusion

In this paper we presented the SEPL framework as a solution to the problem of intra-service protocol specification and execution. SEPL is a DSL whose features to specify service protocols range from basic control flow directives and synchronous/asynchronous invocations to fault handling and easy access to WSRF resource properties or elements in XML markup. Based on the definition of the DSL, we presented the design and implementation of the three main components of the SEPL framework: the UML-based SEPL development facilities, the SEPL execution engine, and the SEPL protocol host (PH). The PH host offers a convenient way to expose protocol functions as Web service operations, thereby shifting the protocol execution responsibility from clients to the service provider or an intermediary. The qualitative evaluation of the framework indicates that SEPL allows efficient development of service protocols and fosters readability and maintainability. The performance evaluation has shown that SEPL protocols execute with a minor overhead compared to static implementation of the protocol logic.

As part of our future work we intend to develop a suitable way to include SEPL protocols in the WSDL definition of services, e.g., linking to a SEPL file or by embedding SEPL code directly. Another possible method to communicate SEPL protocols to service clients would be the use of *WS-MetadataExchange* [2]. Furthermore, we plan to improve the algorithms to generate SEPL code and the protocol WSDL. We also envision alternative ways to identify protocol execution instances, e.g., by means of a conversation identifier in the SOAP header.

9 Appendix: SEPL Syntax Rules in EBNF

```
1  PROTOCOL = {ASSIGNMENT} FUNCTION {FUNCTION} ;
2  FUNCTION = "function" IDENTIFIER FUNC_PARAMETERS BLOCK ;
3  EOL = "\r" | "\n" | "\r\n" ;
4  FUNC_PARAMETERS = "(" [ FUNC_PARAM { "," FUNC_PARAM } ] ")" ;
5  FUNC_PARAM = IDENTIFIER ;
6  BLOCK = BLOCK2 | EXPRESSION {";" [ EXPRESSION ] } ;
7  BLOCK2 = "{" [ EXPRESSION_LIST ] "}" ;
8  EXPRESSION_LIST = EXPRESSION { (";" | EOL) [ EXPRESSION ] } ;
9  EXPRESSION = ASSIGNMENT | STATEMENT_EXPR ;
10 ASSIGNMENT = IDENTIFIER "=" ASSIGNABLE ;
11 ASSIGNABLE = INVOCATION | CONSTRUCTOR | (XML ";") | PRIMARY_EXPR;
12 STATEMENT_EXPR = IF_STMT | WHILE_STATEMENT | DO_STATEMENT |
13                  FOR_STATEMENT | "break" | "continue" | RETURN |
14                  INVOCATION | TRY_STATEMENT | "throw" EXPRESSION ;
15 IF_STMT = "if" "(" EXPRESSION ")" BLOCK {ELSEIF_NODE} [ELSE_NODE] ;
16 ELSEIF_NODE = "else" "if" "(" EXPRESSION ")" BLOCK ;
17 ELSE_NODE = "else" BLOCK ;
18 WHILE_STATEMENT = "while" "(" EXPRESSION ")"  BLOCK ;
19 TRY_STATEMENT = "try" BLOCK2 { CATCH_BLOCK } [ FINALLY_BLOCK ] ;
20 CATCH_BLOCK = "catch" "(" IDENTIFIER ")" BLOCK2 ;
21 FINALLY_BLOCK = "finally" BLOCK2 ;
22 DO_STATEMENT = "do" BLOCK2 "while" "(" EXPRESSION ")" ;
23 FOR_STATEMENT = "for" "(" (IDENTIFIER ":" EXPRESSION) |
24                  ({ASSIGNMENT} ";" [CONDITION] ";" {ASSIGNMENT}) ")"
25                  BLOCK ;
```

```
26 RETURN = "return" [ASSIGNABLE] ;
27 NUMBER = INTEGER | FLOATING_POINT ;
28 DIGIT = "0".."9" ;
29 INTEGER = DIGIT { DIGIT };
30 FLOATING_POINT = INTEGER "." INTEGER [EXPONENT] | INTEGER EXPONENT ;
31 EXPONENT = ("e"|"E") ["+"|"-"] INTEGER ;
32 LETTER = "a".."z" | "A".."Z" | "_" ;
33 IDENTIFIER = LETTER { LETTER | DIGIT } ;
34 NOT_QUOTATION_MARK = "\x0020".."\x0021" | "\x0023".."\xffff" ;
35 STRING = "\"" {NOT_QUOTATION_MARK | "\\\""} "\"" ;
36 STRING_ANY = {NOT_QUOTATION_MARK | "\""} ;
37 NCNAME = IDENTIFIER [ {IDENTIFIER | "-"} IDENTIFIER ] ;
38 XML = "<" [ NCNAME ":" ] NCNAME
39           { [ NCNAME ":" ] NCNAME "=" STRING } ">"
40             (XML | STRING_ANY) "</" [ NCNAME ":" ] NCNAME ">" ;
41 PRIMARY_EXPR = STRING | NUMBER | IDENTIFIER | MATH_EXPRESSION;
42 INVOCATION = IDENTIFIER "(" PARAMETERS ")" ;
43 PARAMETERS = "(" [ PARAM { "," PARAM } ] ")" ;
44 PARAM = IDENTIFIER | STRING | MATH_EXPRESSION ;
45 MATH_EXPRESSION = STRING | NUMBER | IDENTIFIER | UNARY_EXPRESSION |
46                   BINARY_EXPRESSION | "(" MATH_EXPRESSION ")" ;
47 UNARY_EXPRESSION = ("!" | "-" | "~") MATH_EXPRESSION ;
48 BINARY_OPERATOR = ("+"|"-"|"*"|"/"|"%"|"<"|"<="|">"|">="|"||"|"&&") ;
49 BINARY_EXPRESSION = MATH_EXPRESSION BINARY_OPERATOR MATH_EXPRESSION;
50 CONSTRUCTOR = IDENTIFIER "(" PARAMETERS ")" ;
51 CONDITION = MATH_EXPRESSION ;
```

Listing 2 SEPL Syntax Rules in EBNF

Acknowledgements The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement 215483 (S-Cube).

The authors would like to thank the anonymous reviewers for their valuable comments and suggestions for improvement.

References

1. Atkinson, C., Kuhne, T.: Model-driven development: a metamodeling foundation. Software, IEEE **20**(5), 36–41 (2003)
2. Ballinger, K., et al.: Web Services Metadata Exchange (WS-MetadataExchange). http://specs.xmlsoap.org/ws/2004/09/mex/WS-MetadataExchange.pdf (2006)
3. Benatallah, B., Casati, F., Toumani, F.: Representing, analysing and managing web service protocols. Data Knowl. Eng. **58**, 327–357 (2006)
4. Benslimane, D., Dustdar, S., Sheth, A.: Services mashups: The new generation of web applications. IEEE Internet Computing **12**(5), 13–15 (2008)
5. Beran, P.P., Habel, G., Schikuta, E.: SODA A Distributed Data Management Framework for the Internet of Services. In: GCC '08: Proceedings of the 2008 Seventh International Conference on Grid and Cooperative Computing, pp. 292–300. IEEE Computer Society, Washington, DC, USA (2008)
6. Butek, R.: Which style of WSDL should I use? http://www.ibm.com/developerworks/webservices/library/ws-whichwsdl/ (2003). Visited: 2010-02-03
7. D'Agostino, R.B., Stephens, M.A. (eds.): Goodness-of-fit techniques. Marcel Dekker, Inc., New York, NY, USA (1986)
8. Dörnemann, T., Friese, T., Herdt, S., Juhnke, E., Freisleben, B.: Grid Workflow Modelling Using Grid-Specific BPEL Extensions. In: Proceedings of German e-Science Conference 2007, pp. 1–9 (2007)
9. Dustdar, S., Schreiner, W.: A survey on web services composition. International Journal of Web and Grid Services **1**(1), 1–30 (2005)
10. Eclipse Foundation: Model Development Tools (MDT). http://www.eclipse.org/uml2
11. Erl, T.: Service-Oriented Architecture. Concepts, Technology, and Design. Prentice Hall (2005)

12. Ezenwoye, O., Sadjadi, S.M., Cary, A., Robinson, M.: Grid Service Composition in BPEL for Scientific Applications. In: OTM Conferences (2), pp. 1304–1312 (2007)
13. Florescu, D., Grünhagen, A., Kossmann, D.: XL: an XML programming language for Web service specification and composition. Computer Networks **42**(5), 641–660 (2003)
14. Foster, I., Kesselman, C.: The Grid 2: Blueprint for a New Computing Infrastructure. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2003)
15. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1995)
16. Gao, Z., Luo, S., Lin, Y., Ding, D.: A grid-based integration model of heterogeneous database systems. In: Proceedings of the International Conference on Information Technology and Computer Science, pp. 126–129. IEEE Computer Society (2009)
17. Hamadi, R., Benatallah, B.: A Petri net-based model for web service composition. In: ADC '03: Proceedings of the 14th Australasian database conference, pp. 191–200. Australian Computer Society, Inc., Darlinghurst, Australia (2003)
18. Leitner, P., Rosenberg, F., Dustdar, S.: DAIOS - Efficient Dynamic Web Service Invocation. IEEE Internet Computing **13**(3), 72–80 (2009)
19. Lowry, R.: t-Test for the Significance of the Difference between the Means of Two Independent Samples. http://faculty.vassar.edu/lowry/ch11pt1.html. Visited: 2010-03-05
20. Maximilien, E., Ranabahu, A., Gomadam, K.: An Online Platform for Web APIs and Service Mashups. Internet Computing, IEEE **12**(5), 32 –43 (2008)
21. Menasce, D.: Qos issues in web services. Internet Computing, IEEE **6**(6), 72 – 75 (2002)
22. Newcomer, E., Lomow, G.: Understanding SOA with Web Services. Addison Wesley Professional (2004)
23. Object Management Group: OMG Unified Modeling Language (OMG UML), Superstructure, V2.1.2. http://www.omg.org/spec/UML/2.1.2/Superstructure/PDF/
24. Object Management Group: MOF 2.0/XMI Mapping, Version 2.1.1. http://www.omg.org/cgi-bin/apps/doc?formal/07-12-01.pdf (2007)
25. Organization for the Advancement of Structured Information Standards (OASIS): Web Services Base Notification 1.3 (WS-BaseNotification). http://docs.oasis-open.org/wsn/wsn-ws_base_notification-1.3-spec-os.pdf (2006)
26. Organization for the Advancement of Structured Information Standards (OASIS): Web Services Business Process Execution Language Version 2.0. http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html (2006)
27. Organization for the Advancement of Structured Information Standards (OASIS): Web Services Resource Framework. http://www.oasis-open.org/committees/wsrf (2006)
28. Organization for the Advancement of Structured Information Standards (OASIS): Web Services Resource Properties 1.2 (WS-ResourceProperties). http://docs.oasis-open.org/wsrf/wsrf-ws_resource_properties-1.2-spec-os.pdf (2006)
29. Papazoglou, M.P., Traverso, P., Dustdar, S., Leymann, F.: Service-Oriented Computing: State of the Art and Research Challenges. Computer **40**(11), 38–45 (2007)
30. Parastatidis, S., Woodman, S., Webber, J., Kuo, D., Greenfield, P.: Asynchronous Messaging between Web Services Using SSDL. IEEE Internet Computing **10**(1), 26–39 (2006)
31. Peltz, C.: Web services orchestration and choreography. Computer **36**, 46–52 (2003)
32. Puhlmann, F., Weske, M.: Using the *pi*-Calculus for Formalizing Workflow Patterns. In: Business Process Management, pp. 153–168 (2005)
33. Ryu, S.H., Saint-Paul, R., Benatallah, B., Casati, F.: A Framework for Managing the Evolution of Business Protocols in Web Services. In: Proceedings of the fourth Asia-Pacific Conference on Conceptual Modelling (APCCM'07), pp. 49–59 (2007)
34. Strembeck, M., Zdun, U.: An approach for the systematic development of domain-specific languages. Softw. Pract. Exper. **39**(15), 1253–1292 (2009)
35. Sun Microsystems Inc.: JSR-000154 JavaTM Servlet 2.4 Specification. http://jcp.org/aboutJava/communityprocess/final/jsr154/
36. (W3C), W.W.W.C.: Web Services Conversation Language (WSCL) 1.0. http://www.w3.org/TR/wscl10/ (2002)
37. Zhang, J., Chang, C.K., Chung, J.Y., Kim, S.W.: WS-Net: a Petri-net based specification model for Web services. In: Proceedings of the IEEE International Conference on Web Services (ICWS) (2004)
38. Zhu, F., Turner, M., Kotsiopoulos, I., Bennett, K., Russell, M., Budgen, D., Brereton, P., Keane, J., Layzell, P., Rigby, M., Xu, J.: Dynamic data integration using web services. In: ICWS '04: Proceedings of the IEEE International Conference on Web Services, p. 262. IEEE Computer Society, Washington, DC, USA (2004)