FAKULTÄT FÜR !NFORMATIK

Towards a Domain-Specific Language for Defining Intra-Service Protocols of Stateful Web Services

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Waldemar Hummer
Matrikelnummer 0416710

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung:
Betreuer: Univ.Prof. Dr. Schahram Dustdar
Mitwirkung: Univ.Ass. Mag. Philipp Leitner

Wien, 27.03.2009 (Unterschrift Verfasser) (Unterschrift Betreuer)

Erklärung zur Verfassung der Arbeit

Waldemar Hummer
Leidesdorfgasse 11-13/3/2
1190 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 27. März 2009

_____
(Unterschrift)

Abstract

The *Web Services Resource Framework* (WSRF) specifications are an extension to the core Web service standards (WSDL, SOAP, UDDI), which define how to use Web service technologies to address and access computational resources that maintain an internal state between invocations. This behavior is referred to as *stateful*.

Web services which address stateful resources usually have implicit constraints concerning the order in which operations can be invoked. For example, a client might first have to invoke an initialization method before further invocations to the stateful resource are allowed. In order for the client to adhere to these constraints, it needs to obtain information about the *intra-service protocol* in terms of 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 thesis we introduce *SEPL* (*SErvice Protocol Language*) as a high-level domain-specific language (DSL) with a scripting-language like syntax, that is tailored specifically towards the needs of intra-service protocol description. 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 can use SEPL to define the procedure which must be followed by clients in order to achieve a certain behavior. These functionalities, which are compounded from a combination of the service's operations, are themselves exposed as stand-alone operations. Essentially, this approach can break down the functionalities which require knowledge about the state of stateful services to a "stateless" interface. We provide a graphical representation of service protocols in the form of *UML Activity Diagrams* and tools to generate SEPL code from such models. The broad range of existing UML design tools can foster model driven development of SEPL service protocols. We further present a solution to host and execute SEPL protocols in a server application based on Web services technology.

Kurzfassung

Das *Web Services Resource Framework* (WSRF) ist eine Sammlung von Spezifikationen zur Erweiterung der fundamentalen Web Service Standards (WSDL, SOAP, UDDI), die beschreibt, wie Web Service Technologien verwendet werden, um Computer und elektronische Einrichtungen (*Ressourcen*) zu adressieren und aufzurufen, die einen internen Zustand zwischen aufeinanderfolgenden Aufrufen beibehalten. Diese Eigenschaft wird als *stateful* (engl. für "zustandsbehaftet") bezeichnet.

Web Services, die auf zustandsbehafteten Ressourcen aufbauen, erwarten in der Regel eine bestimmte Reihenfolge der Operations-Aufrufe. Zum Beispiel könnte es vom Client erwartet werden, zuerst eine Initialisierungs-Methode aufzurufen, bevor weitere Anfragen an die zustandsbehafteten Ressource erlaubt sind. Damit der Client sich an diese Erwartungen und Einschränkungen halten kann, muss er Informationen über das Service-interne Protokoll (*intra-service protocol*) erhalten. Dies betrifft die erlaubten Operations-Sequenzen und die Transformation von Input und Output, die zwischen den Aufrufen erfolgen muss. Während sich Anbieter und Anwender weitgehend auf WSDL als Standard zur Beschreibung des statischen, funktionalen Interfaces von Web Services geeignet haben, existiert noch ein offensichtlicher Mangel an Sprachen um das dynamische, verhaltensbezogene Interface von Services niederzulegen.

In der vorliegenden Master-Arbeit führen wir die auf hoher Abstraktionsebene angesiedelte domänenspezifische Sprache SEPL (*SErvice Protocol Language*) ein, die mit ihrer Skriptingsprachen-ähnlichen Syntax auf die Anforderungen der Beschreibung von Service-internen Protokollen zugeschnitten ist. Namhafte Features der Sprache sind beispielsweise die Unterstützung von WS-Addressing und die einfache Erzeugung von neuen Web Service Instanzen, synchrone und asynchrone Möglichkeiten des Aufrufs von Service-Operationen sowie vereinfachter Zugriff auf Zustandsvariablen im Stil von WSRF (*resource properties*). Service-Provider verwenden SEPL, um die Prozedur zu definieren, an die sich die Clients halten müssen, um ein gewisses Verhalten zu erreichen. Diese Funktionalitäten, die sich aus einer Kombination von Aufrufen der einzelnen Operationen des Services ergeben, werden ihrerseits als selbständige Operationen angeboten. Durch diesen Ansatz ist es möglich, die Funktionalitäten, die ein Wissen über den Zustand eines Services erfordern, auf ein zustandsloses (*stateless*) Interface herunterzubrechen. Im Zuge der Arbeit wird eine graphische Repräsentation von SEPL Dokumenten in der Form von UML Aktivitätsdiagrammen (*activity diagrams*) vorgestellt, sowie ein Werkzeug angeboten, das SEPL Code aus solchen UML Modellen generiert. Die breite Palette an vorhandenen UML Design-Tools begünstigt die modellgetriebene Entwicklung von SEPL Protokollen. Des Weiteren präsentieren wir eine Lösung zum Hosten von SEPL Protokollen in einer Server-Anwendung, die als Web Service erreichbar ist und die Protokoll-Ausführung vornimmt.

Danksagungen

Die vorliegende *Master's Thesis* stellt die Krönung von fünf arbeits-, erfolg- und ereignisreichen vorangegangenen Studienjahren dar. Die Fertigstellung und Einreichung der Arbeit ist zum einen ein erbaulicher Moment, wenn man an die Strapazen und Belastungen sowie den immanenten Schlafentzugs-Charakter der vergangenen Jahre denkt. Zum anderen stimmt einen der drohende Verlust der Freiheit nachdenklich: eine derart selbstbestimmte Art der Zeiteinteilung ist für das (reine) Berufsleben in Zukunft wohl nicht zu erwarten. Für mich steht jedenfalls fest: die Studienjahre sind eine "goldene Ära" im individuellen Werdegang einer Person. So würde ich mir aus heutiger Sicht – wie viele andere auch – wünschen, dass sie ewig andauern mögen.

Mein Dank für die Ermöglichung der wunderbaren Studienzeit richtet sich in erster Linie an meine lieben Eltern, die mir den Weg zu dieser unversitären Ausbildung geebnet haben. Ohne Eure finanzielle, intellektuelle, motivatorische und moralische Unterstützung würde ich heute nicht "da stehen, wo ich mich befinde" (und das Vergnügen haben, diese Danksagung zu schreiben). Ebenso muss ich meinen lieben Schwestern danken, die immer ein aufmunterndes Wort für mich übrig hatten, wenn der Druck besonders groß war. Nicht zuletzt gilt mein Dank allen Verwandten, Bekannten, Freundinnen und Freunden, von denen jeder und jede einzelne auf wichtige Weise meinen Weg mitgestaltet hat.

Für die inhaltliche Unterstützung bei der Erstellung dieser Arbeit möchte ich mich herzlich bei meinen Betreuern, Prof. Dr. Schahram Dustdar und Mag. Philipp Leitner, bedanken. Die Atmosphäre am Institut war überaus angenehm, auf Feedback musste ich nie lange warten und mir wurde in weiten Teilen Freiheit geboten, sodass ich mich weder über- noch unterbetreut gefühlt habe. Die Besprechungen und Treffen waren fachlich und persönlich hochgradig ergiebig, weshalb ich das Verfassen der Arbeit nicht als Last, sondern viel mehr als Bereicherung empfunden habe.

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

Throughout the last years, software engineering research and practice have put remarkable focus on the **Service-Oriented Architecture** (SOA) [16, 28, 41] paradigm, which propagates the use of *services* - autonomous applications made available in a computer network using standardized interface description and message exchange - as a means to create decoupled, distributed, composite applications in heterogeneous environments. *Web services* [76] have gained momentum as a means for implementing SOA applications. The technologies and standards for Web services have been widely agreed on: the Simple Object Access Protocol (SOAP) [78, 92] defines standards for the exchange of messages between client applications and service endpoints; the Web Services Definition Language (WSDL) [81] is used to provide metadata about a service, i.e., under which endpoint address it is available, which operations it offers and which messaging technology it supports; Web Services Addressing (WSA) [84] provides an extensible mechanism to uniquely address Web service endpoints; XML [74] is the syntactical foundation for SOAP and WSDL as well as many other standards.

It is a commonly agreed principle that Web services generally do not persist a state across invocations, i.e., are **stateless** [16, 41]. The output and side effects of invocations to stateless Web services depend solely on the input, the service business logic and environmental factors (e.g., a back-end application, an attached machine, the current time, ...) but not on the state of the service itself. This implies that two service invocations with the same input and under the same environmental conditions should result in the same output and side effects. However, in some areas **stateful** services have become popular. Most notable is the concept of the *Grid service* which is defined in [20] 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) "*. More information on Grid computing [20] and Grid services will be given in Section 2. Participants in stateful service interactions are

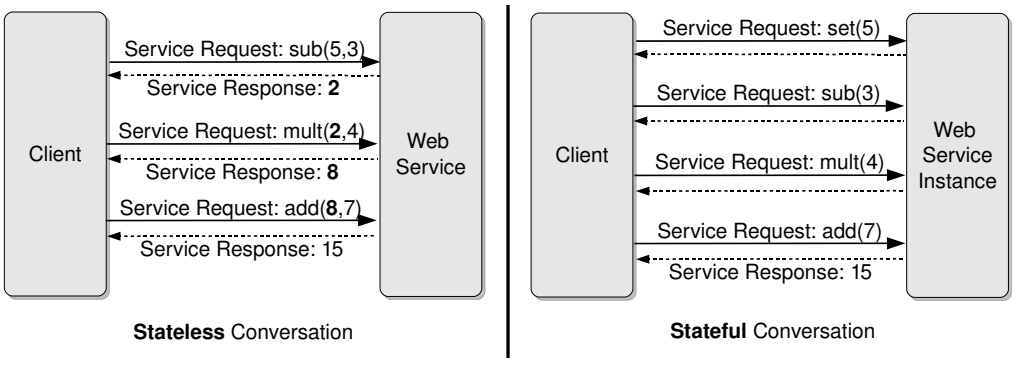


Figure 1: Stateless Versus Stateful Service Conversation

service *instances*, which maintain an internal state. Figure 1 illustrates the difference between stateful and stateless conversation by means of two *calculator service* im-

plementations and clients which calculate the result of the expression $(5-3)*4+7$ using the services' operations. In the stateless conversation, firstly the operation `sub` with parameters `5` and `3` is invoked, then the result (`2`) is passed to the operation `mult` along with the value `4` and so on: the result of each invocation is passed as a parameter to the next operation. In the stateful conversation, the Web service instance internally maintains a *current value*. The first invocation sets the value to `5`, the second subtracts `3` from the value, the third multiplies the value with `4` and the fourth adds `7` to the value. Note that the intermediate results do not have to be passed to the subsequent operations and that only the return value of the last invocation is needed (final result). Since less data needs to be transmitted over the network, the Grid service paradigm is interesting for scientific, technical or medical applications, which access large amounts of data.

In practice, the capabilities of one Web service often only account for a small part of the desired functionality and hence services are combined. In cases where the execution of a service operation involves the invocation of one or more other service operations, we speak of **service composition** [13]. According to the survey carried out in [13], service composition approaches can be divided into various categories, which are briefly summarized below:

- The involved services can be assembled *statically* at design-time or *dynamically* at runtime. Static compositions, on the one hand, usually have a performance advantage but on the other hand require adaption if the services evolve. The dynamic approach aims at making use of the flexibility of the service environment by assembling the required partner services upon request – based on given requirements. In compliance with the SOA triangle [41], service providers publish services and their capabilities in a registry. Then the composition engine, based on the user requirements, requests description documents of appropriate services and binds to these concrete services.
- In *manual* service composition, participating services are selected by humans and statically linked to one another. On the other hand, *automated* service composition aims at taking away this responsibility by leaving to choice to "intelligent" decision algorithms, which automatically select suitable services to satisfy the requirements of a given abstract composition problem.
- In *model driven* service composition, the Unified Modeling Language [46] (UML) is used to provide a high-level representation of services with the aim of enabling a direct mapping to concrete service composition languages [13]. By means of the *Object Constraint Language* [47] (OCL) it becomes possible to specify business rules and constraints which apply to the service operations and interdependencies. The authors of [60] identify seven main entities which apply to all descriptions of service compositions: *activity* (a well-defined business function), *condition* (integrity constraints and guards for activities), *event* (occurrences of normal and exceptional nature), *flow* (block of activities and how they are connected), *message* (container of information; input and output of activities),

provider (participating party) and *role* (abstract description of a provider).

- Client requests to a composition engine might be expressed in a *declarative* way using formal languages: the engine is then responsible to firstly construct a generic plan, and, based on that, to discover appropriate services and build a workflow out of them [13].

Based on the granularity of the composition we distinguish *microflows* (*"short running, more technical processes")* and *macroflows* (*"long-running, higher-level business process"*) [26]. This conception is illustrated in Figure 2. The macroflow printed in a gray rectangle embraces three activities, which make up a higher-level business process. Two of these activities (`Activity 1` and `Activity 2`) are processes themselves

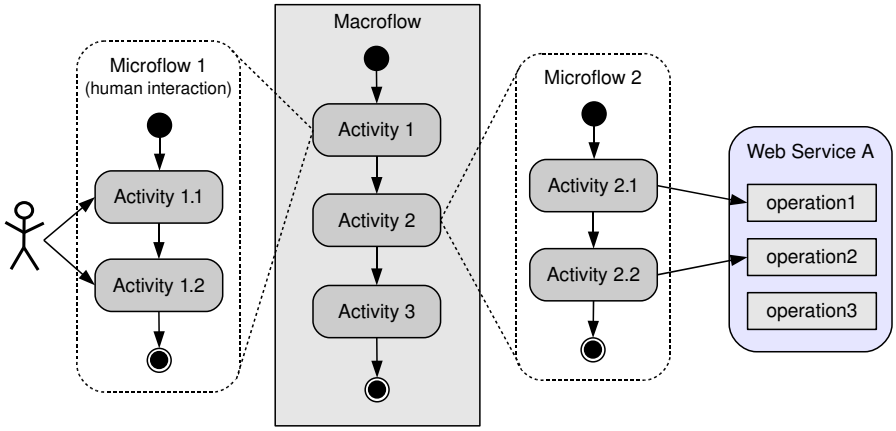


Figure 2: Macroflows and Microflows

– on a microflow level. `Microflow 1` includes human interaction whereas `Microflow 2` is executed programmatically. `Microflow 2` invokes operations of a Web Service `A` in a certain order and manner. The microflow is an invokable sub-process which executes autonomously: the internal procedures are hidden (and irrelevant) on the macroflow level. Microflows have transactional characteristics, i.e., they require to perform all included activities successfully in order to provide their functionality to the parent macroflow.

1.1 Motivation

Clients need to obtain information about a service in order 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 covered by the WSDL document offered by the service provider. In the WSDL document, client applications determine the operation names in the `portType` section, parameter and return types in the `portType` and `types` sections and the message style in the `binding` section. On the other hand, clients ought to know the service's

"dynamic" (behavioral) interface which specifies the order in which operations can be invoked. Especially *stateful* services, i.e., services which persist data values across invocations, often provide functionalities which involve more than one operation.

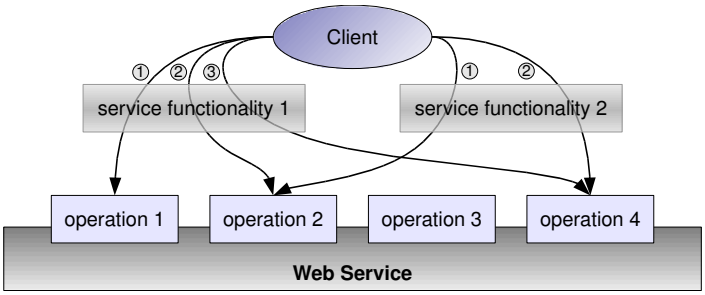


Figure 3: Service Functionalities Involving Service Operations

Figure 3 depicts a scenario where a Web service, as for its "static" interface, offers 4 operations. Assuming that the client wants to invoke `operation 4` and that the business logic in that operation depends on the former execution of `operation 2`, the client would first have to invoke `operation 2` before it invokes `operation 4`. In Figure 3 this *sequence* of service invocations is named "service functionality 2". "Service functionality 1" requires that `operation 1`, `operation 2` and `operation 4` shall be invoked (in this order). 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*. We observe that intra-service protocols (for short, service protocols) relate to service microflows:

- Both service protocols and microflows create new or extended functionality in addition to the single operations offered by a Web service.
- Service protocols – like microflows – have a transactional aspect: if one involved operation call fails, the wanted functionality can most likely not be delivered.
- Service protocols target one single service and they contain no cross-service collaboration aspects. This is equally true for microflows which operate at the lowermost level.
- Service protocol descriptions are actually a means of expressing the set of microflow processes which are supported by a specific Web service ("dynamic interface").

Before continuing, we summarize the key terms mentioned so far in Table 1.

In real-world applications it is very likely that a service contains constraints concerning valid and invalid operation sequences, which becomes clear when considering the following examples:

- A service implements the *state pattern* [21] (or, alternatively, the *strategy pattern*) and reacts to client requests according to its internal state. The client

Term	Description
Stateful Service	A stateful service is a service which maintains state information across invocations. For the interaction with stateful services it is crucial for clients to know the protocol that has to be stuck to. Stateful services play an important role, e.g., in Grid computing. The WSRF [58] set of specifications enables explicit addressing and handling of stateful services.
Static Interface	The static service interface describes which operations are supported and which input and output is expected. For Web services, it is expressed in WSDL.
Intra-Service Protocol = Dynamic Interface	The intra-service protocol defines behavioral aspects of services on the macroflow level: in which order and with which output-input transformations operations have to be invoked in order to achieve certain functionalities.
Service Composition	Service composition is the process of creating new invokable services which involve the functionality of existing services. In contrast to intra-service protocols, service composition usually involves several partner services ("inter-service") and therefore covers aspects of the macroflow level.

Table 1: Table of Definitions

sets the desired state of the service (by invoking an according operation) before invoking any business logic operations.

- The input of an operation requires the result (output) of another operation.
- A client invokes a service operation which creates a new service *resource* [58] and returns the *endpoint reference* of the new resource. The client needs to invoke an initialization operation on the newly created resource. This architectural pattern is called the *resource factory pattern* [22] and will be further discussed in Section 2.
- A service with 2 or more operations is to be extended by a new functionality which involves invoking the existing operations. The WSDL contract cannot be extended by a new operation because then some existing clients of the service might fail to parse it.

Currently, there is still an evident lack of special purpose languages to describe intra-service protocols. In our state of the art review in Section 2 we will discuss the related problem of service *composition*, which aims at composing a set of services to an invokable business process. Service composition has been addressed in various ways and the suggested solutions range from semantics-/ontology-based approaches to process modeling using Petri nets [25] or finite state machines [4, 30]. The most prominent de facto standard language for service composition is *WS-BPEL* [56], an XML-based language used to create invokable electronic business processes.

Describing intra-service protocols is essentially a subproblem of service composition specification, therefore, languages from the service composition domain (e.g. WS-BPEL) can also be used to specify intra-service protocols. However, we argue that this approach has a number of drawbacks. Firstly, the problem of intra-service protocol specification is less complex than service compositions: the service to invoke is always clearly defined (e.g. there is no need for partner links), protocols are usually much less complicated than composition, and many WS-BPEL constructs (e.g. "Parallel Flows") are not useful for intra-service protocols. Secondly, composition languages are generic and do not contain any explicit support for stateful Web services specifics, such as *Web Service Resource Framework* (WSRF) [58] Resource Properties (WS-ResourceProperties) [59] or the WSRF factory/instance pattern. Thirdly, composition engines are rather heavy-weight server tools, and not suitable for client-side usage. Lastly, the XML syntax that, e.g., WS-BPEL is based on is notoriously hard to write without appropriate tool support.

1.2 Contribution

In this thesis we address the issues mentioned before and propose a solution for the problem of intra-service protocol modeling, description and execution. The contribution of this thesis is threefold:

- We introduce a light-weight, scripting-like 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 to interpret for clients. In Section 4 we present our prototype SEPL execution engine and SEPL client implementation written in the Java programming language.
- We provide a framework for the **model driven development** [3] (MDD) **of SEPL documents** using the Unified Modeling Language (UML). For this purpose we define a 1-to-1 mapping from UML activity diagrams [44] to SEPL code. Among the various graphical UML modeling tools available we decided to choose the *Eclipse Model Development Tools* (MDT) [15] platform as the starting point for SEPL modeling. MDT contains a graphical editor to compose UML activity diagrams and save them in an XML format based on *XML Metadata Interchange* [48] (XMI). Based on these files we implemented a command-line SEPL code generation tool *UML2SEPL*.
- To take away the responsibility of clients to execute SEPL protocols, we offer a **SEPL protocol server** implementation. The responsibility of the server is to host SEPL protocols, to expose service functionalities contained therein as

WSDL operations and to execute protocol functions upon request. We implemented the SEPL protocol server as a configurable Web Application Archive (WAR) [71] that can be deployed to established application servers like Apache's Tomcat [2] or Sun's Glassfish [8].

1.3 Organization

The remainder of this thesis is structured as follows:

- Section 2 details the current state of the art in the areas of SOA, stateless and stateful Web services, service composition and service protocols. It presents the *Web Services Resource Framework* (WSRF), a set of specifications to handle stateful Web services and service resources. Furthermore we briefly discuss *WS-Addressing* [84], *WS-BaseNotification* [55] and *SOAP Fault* [78].
- Section 3 provides an overview of related work in the area of intra-service protocol description and the related field of service composition. We examine the concept of *conversations* in the message exchange with services and, relatedly, the *Web Services Conversation Language* [80]. We discuss approaches to *Petri Net* [67] based and model-driven development of service compositions. Furthermore, we review *SSDL* [66], a SOAP-centric, message-based Web service description language which attempts to avert the shortcomings of WSDL and draws a complete picture about services and their protocols.
- In Section 4 we firstly define SEPL, a domain-specific language which addresses the problem of intra-service protocol definition. Afterwards we discuss the design of the prototypical SEPL execution engine which has been developed as part of the practical work of this thesis. Secondly, we explain how intra-service protocols can be modeled using UML activity diagrams and describe the implementation of our "UML-to-SEPL" code generator. Thirdly, we address the problem of how service protocols can be hosted in a server to expose the higher-level service functionalities as Web service operations themselves.
- Section 5 covers a presentation of the prototype implementation of the SEPL execution engine, the SEPL code generator and the SEPL service host. Selected aspects such as the SEPL code writer algorithm and the WSDL generation algorithm are discussed in more detail. Usage examples illustrate how the respective components are integrated into new applications.
- Section 6 contains an evaluation of the implemented SEPL framework concerning different aspects. The productivity of developing service protocols with SEPL is measured against other methods (pure client implementation, implementation in WS-BPEL). The section also covers a performance evaluation of the protocol execution and the WSDL generation, which are crucial parts of the SEPL prototype implementation.
- Section 7 concludes the thesis with a short summary and plans for improving the framework and work to be carried out in the future.

2 State of the Art Review

In this section we discuss the concepts of *Service Oriented Architecture*, the core protocols and specifications of *Web services* and the current state of the art in Web service composition.

2.1 Service Oriented Architecture

Since software development is a complex, time consuming and expensive process, vendors as well as customers can benefit from *reusable* software elements. To achieve sustainable reuseability, on a Business-internal as well as on a Business-to-Business (B2B) level, a high degree of *interoperability* and integration is crucial. The concept of *Service-Oriented Computing* [61] (SOC) utilizes *services* as fundamental elements for developing applications: services are *"self-describing, platform-agnostic computational elements that support rapid, low-cost composition of distributed applications"* [61]. According to [16], characteristics of services are:

- *Loose coupling*: Services are autonomous and not hard-wired. The relationship between services minimizes dependencies and allows for the replacement of single elements.
- *Service contract*: Services adhere to a communications and interface agreement, as defined by one or more service description documents.
- *Autonomy*: Services have control over the logic they realize.
- *Abstraction*: Services describe their interface in a service contract but hide the actual implementation logic from the outside world.
- *Reuseability*: The functionality provided by a service is intended for reuse.
- *Composability*: Services can be assembled to form composite services. The *abstraction* principle still applies, which means that to the outside world composite services are not distinguishable from atomic services.
- *Statelessness*: Services do not retain an internal state. Context information is carried in the message exchange with the service.
- *Discoverability*: Services are designed to be outwardly descriptive so that they can be found and assessed via available discovery mechanisms.

The view of *software as a service* is the basis for the *Service Oriented Architecture* (SOA) paradigm, a form of technology architecture that adheres to the principles of service-orientation [16]. The definition given in [41] underlines the interoperability and platform-independence aspects of SOA:

"A service-oriented architecture is a style of design that guides all aspects of creating and using business services throughout their lifecycle (from conception to retirement). An SOA is also a way to define and provision an IT infrastructure to allow different

applications to exchange data and participate in business processes, regardless of the operating systems or programming languages underlying those applications."[41]

SOA is best described by means of the SOA "triangle" [41], which depicts the dependencies between the three roles of *Service Provider*, *Service Consumer* and *Service Registry* (see Figure 4). The service consumer requires a certain capability and is

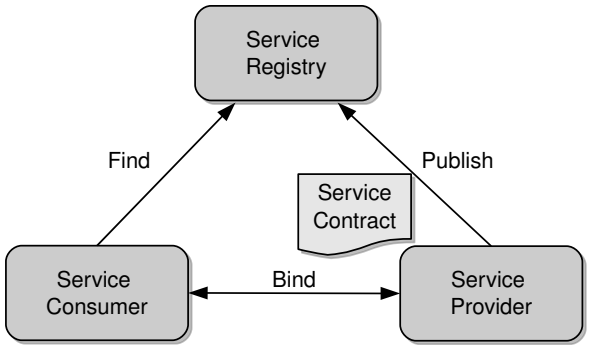


Figure 4: Roles and Relationships in the SOA Triangle

looking for a service which can provide it. The service provider is able to provide the required capability in the form of a service. The service registry is the mediator between consumer and provider: the service provider *publishes* the service contract to the registry; the consumer queries the service registry to *find* the desired service and its contract. When the service consumer has found an appropriate service, the *binding* takes place and interaction between the consumer and the provider (service) begins.

2.2 Web Services

One possible – and, by this time, very popular – way to implement SOA applications is by means of *Web services* [11, 41, 61, 76]. [1] Web service technology builds the bridge from the abstract SOA concepts to concrete service implementations. Essentially, the term Web service embraces a set of standards and specifications for different aspects of SOC. Three core specifications are briefly presented in the following:

- *Description of Web services*: The Web Services Definition Language [81] (**WSDL**) serves the purpose of defining the service contract in the sense of the service's interface. For one thing, WSDL describes which *operations* the service offers, which input and output *messages* the operations receive and return and which exact XML schema *type* [86, 87] these messages have to match. This information makes up the `portType` element of WSDL documents and constitutes the abstract interface without any dictate of the transportation method to be used.

[1] Note that it is a common misperception that all applications using Web services are service-oriented or that Web services are the only way of creating SOAs [16].

Based on the `portType`, WSDL documents contain a `binding` section, which determines the transport protocol to be used (usually HTTP [18]). When the WSDL declares *SOAP* [78] (see next point) to be the messaging protocol, this section contains specific data such as the *SOAP binding style* [6] or `SOAPAction` strings. The `service` section, finally, specifies the location URI [5] under which the service is accessible.

- *Message exchange with Web services*: The *Simple Object Access Protocol* [78] (**SOAP**) is the most commonly used messaging protocol in Web service computing. SOAP messages are XML documents which are exchanged by service consumer and provider for the invocation of service operations and their return values. The top-level element in SOAP 1.1 messages is named `Envelope` and contains a mandatory `Body` element and an optional `Header` element (in the meantime, SOAP version 1.2 [92] has been released but not yet fully adopted by frameworks and toolkits, so we will consider SOAP version 1.1 throughout this document). The format of SOAP documents follows a separation-of-concerns principle: the `Body` carries the actual payload of the communication, whereas the `Header` contains contextual information to satisfy issues such as addressing (WS-Addressing [84], see below), security (WS-Security [52]) or policy assertions (WS-Policy [90]).
- *Web services registries*: *Universal Description, Discovery and Integration* [73] (**UDDI**), an OASIS specification for Web service registry implementations, defines a data model to describe service providers, their offered services and relations within the entities. UDDI registries describe services verbally (informal) as well as technically (formal) and can be used by consumers via special query operations. Technical service descriptions usually link to the service's WSDL definition.

From the Web services perspective, the SOA triangle has not been fully realized due to the weak adoption of UDDI [16, 38] (the authors of [38] even write about the "*broken* SOA triangle"). In practice, it is more common to use in-house registry implementations and, relatedly, service *factories* (see later). A more complete picture of the so-called Web services *stack* [23, 79] is given in Figure 5. The *network transport*

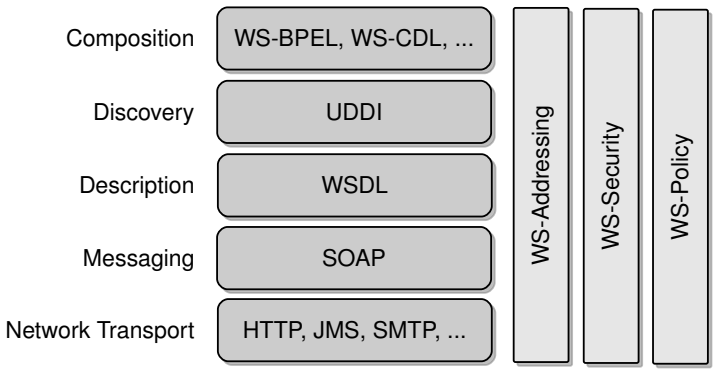


Figure 5: Specifications in the Web Services Stack

protocol (mostly HTTP) is the basis for *messaging* (exchange of SOAP messages). One (meta-)level higher we find WSDL for the *description* of the service interface and expected communication methods. Service *discovery* (UDDI) makes use of service descriptions and the other aforementioned levels and, finally, Web service *composition* involves all other levels and represents the top of the stack. Namable standards on this level are WS-BPEL [56] and WS-CDL [96]. Web service composition will be discussed in more detail later on.

2.3 Stateful Web Services and Service Resources

In the SOA literature of the past years, (Web) services have frequently been characterized as being inherently *stateless* [16, 41] ([41] denotes this a *secondary characteristic* for Web services). The service offers a WSDL contract to its clients and the information contained therein is sufficient to invoke the service's operations. The service implementation does not maintain a state and for the same input an operation will always return the same, or a *logically conforming* result (output) in subsequent invocations. We use the term "logically conforming" because an operation might still react differently depending on externally influencing factors like, for example, the *date and time* of the invocation, but it does not behave dependent on the data stored in the service implementation itself. Paradigms like the *state pattern* can not be implemented when strictly sticking to the principle that services are stateless. In this section we discuss state-of-the-art Web service technologies adverse to the "stateless paradigm".

2.3.1 Grid Computing and the Open Grid Services Infrastructure

Disaccording to the principle of stateless services, Web service developers turned out to be relying on *stateful* services in some areas. Most notably, stateful services play a decisive role in *Grid Computing*. A definition of the "Grid" can be found in [20]: *"We define a Grid as a system that coordinates distributed [computational] resources using standard, open, general-purpose protocols and interfaces to deliver nontrivial qualities of service."*. The coordination of distributed resources requires introspection functions for relevant state information such as current load, availability and response time. In 2003, the *Open Grid Forum* released the specification *Open Grid Services Infrastructure* (OGSI) [49], which satisfies the requirements of Web services for use in a Grid architecture. The theoretical background of OGSI is elaborated in the *Open Grid Services Architecture* [50] (OGSA) specification. OGSA aims to create interoperable, portable and reusable components and systems by defining a set of capabilities and behaviors that address key concerns in Grid systems - discovery, access, allocation, monitoring etc. [50]. A complete description of OGSA would go beyond the scope of this thesis, so we only focus on OGSI and the parts which are relevant in our context.

OGSI defines a component model that extends WSDL and XML Schema definition to incorporate the concepts of stateful Web services. A main facet of handling stateful services with OGSI is that it exposes a service instance's state data for query and update operations as well as change notifications. The state data of a service consists of one or more *service data elements* (SDEs), which are comparable to member variables of a class in object oriented programming. Analogously, the set of all SDEs is called the *service data*, or `serviceData`. Such `serviceData` is defined in the service's WSDL document using so-called Grid WSDL Extensions. Listing 1 depicts an example WSDL document which contains a Grid WSDL `portType` named `gridServicePT`. In addition to the operation declarations this `portType` defines two SDEs, `data1` and `data2`. The purpose of the `staticServiceDataValues` element is to assign initial values. Since the sole way of communication with a Web service means invoking one of its operations, OGSI services must provide methods to query and update service data.

```
1  <wsdl:definitions
2   xmlns:sd="http://www.gridforum.org/namespaces/2003/03/serviceData"
3   xmlns:gwsdl="http://www.gridforum.org/
4           namespaces/2003/03/gridWSDLExtensions"
5   xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
6   xmlns:tns="xxx" targetNamespace="xxx">
7   <gwsdl:portType name="gridServicePT">
8    <wsdl:operation name="..."> ... </wsdl:operation>
9    ...
10   <sd:serviceData name="data1" type="xsd:String" />
11   <sd:serviceData name="data2" type="tns:SomeComplexType" />
12   <sd:staticServiceDataValues>
13     <tns:data1>initial value</tns:data1>
14   </sd:staticServiceDataValues>
15  </gwsdl:portType>
16  ...
17 </wsdl:definitions>
```

Listing 1: Example of Grid WSDL Extensions

2.3.2 The Web Services Resource Framework

The consequence was the need for a unified description of stateful services which resulted in the creation of the *Web Services Resource Framework* (WS-Resource Framework, WSRF) [58] set of specifications. In its introductory document, the authors define the term *WS-Resource* as *"the composition of a [computational] resource and a Web service through which the resource can be accessed"* [57]. Every WS-Resource is represented by an endpoint reference (EPR), which is an XML element whose type is defined by the WS-Addressing specification [84] (see Subsection 2.3.3).

WS-ResourceProperties (WSRP) [59] describes how stateful service resources can expose *properties* and how the definition of these properties becomes a part of the service's WSDL contract. Listing 2 contains an example taken from the specification, which shows the WSDL contract of a "generic disk drive" service.

```
1  <wsdl:definitions ... xmlns:tns="http://example.com/diskDrive" ...>
2   ...
3    <wsdl:types>
4     <xsd:schema targetNamespace="http://example.com/diskDrive" ... >
5     <!-- Resource property element declarations -->
6     <xsd:element name="NumberOfBlocks" type="xsd:integer"/>
7     <xsd:element name="BlockSize" type="xsd:integer" />
8     <xsd:element name="Manufacturer" type="xsd:string" />
9     <xsd:element name="StorageCapability" type="xsd:string" />
10    <!-- Resource properties document declaration -->
11    <xsd:element name="GenericDiskDriveProperties">
12      <xsd:complexType>
13        <xsd:sequence>
14          <xsd:element ref="tns:NumberOfBlocks"/>
15          <xsd:element ref="tns:BlockSize" />
16          <xsd:element ref="tns:Manufacturer" />
17          <xsd:any minOccurs="0" maxOccurs="unbounded" />
18          <xsd:element ref="tns:StorageCapability"
19                        minOccurs="0" maxOccurs="unbounded" />
20        </xsd:sequence>
21      </xsd:complexType>
22    </xsd:element>
23   ...
24    </xsd:schema>
25   </wsdl:types>
26   ...
27   <!-- Association of resource properties document to a portType -->
28   <wsdl:portType name="GenericDiskDrive"
29    wsrf-rp:ResourceProperties="tns:GenericDiskDriveProperties" >
30    <operation name="start" .../>
31    <operation name="stop" .../>
32   ...
33   </wsdl:portType>
34   ...
35  </wsdl:definitions>
```

Listing 2: Resource Property Definitions in WSDL. From: WSRP specification

The `types` section contains an XSD element `GenericDiskDriveProperties`, which itself defines the properties which apply to every instance (WS-Resource) of the service: `NumberOfBlocks`, `BlockSize`, `Manufacturer` and `StorageCapability`. Using the attribute `wsrf-rp:ResourceProperties`, the XSD definition of the properties is tied to the port type `GenericDiskDrive`. Hence, clients can discover available properties and read, change and query resource properties. The respective predefined operations are named `GetResourceProperties`, `SetResourceProperties` and `QueryResourceProperties`.

The explicit addressing of Web service instances (WS-Resources) allows for the dynamic run-time instantiation of new service EPRs. The *WS-Resource Factory Pattern* [22] embodies the concept of a singleton service which - upon request - creates new instances of a specific service and returns its EPR. Due to the fact that service factories are a diverse topic and hard to reduce to a common denominator, there is no explicit standardization in the WSRF specification, but it certainly is common practice to use service factories.

2.3.3 Web Services Addressing

The *Web Services Addressing* specification (WS-Addressing, WSA) provides *"transport-neutral mechanisms to address Web services and messages"* [84]. Before the introduction of WSA, the addressing of a Web service was often dependent on the used transport protocol. In most Web service environments the *Hypertext Transfer Protocol* (HTTP) [18] serves as the transport protocol for SOAP messages. The central addressing entity of HTTP is the Uniform Resource Locator (URL). Service URLs on HTTP level are usually `http://<host>:<port>/<service path>`. With WSA, the addressing information is lifted one level higher in the Web services stack: it is included in the SOAP header of messages. The central XML data structure in WSA is the *endpoint reference* (EPR), which uniquely addresses a Web service or WS-Resource.

```
1 <wsa:EndpointReference>
2    <wsa:Address>http://company.com/ShoppingService</wsa:Address>
3    <wsa:ReferenceProperties>
4       <customerID>customer1529</customerID>
5    </wsa:ReferenceProperties>
6    <wsa:ReferenceParameters>
7       <shoppingCartID>37542</shoppingCartID>
8    </wsa:ReferenceParameters>
9 </wsa:EndpointReference>
```

Listing 3: WS-Addressing EndpointReference Type

Listing 3 shows an example EPR which addresses the WS-Resource available under the address `http://company.com/ShoppingService`, whose property `customerID` equals the value `customer1529`. The contextual information `shoppingCartID` is considered a *parameter* specific to the current interaction with the WS-Resource. The `ReferenceProperties` element is an important part of the EPR for WS-Resources. Consider a factory Web service F which creates service instances $I_1, I_2, \ldots, I_n$. Then, the factory service provides for distinguishable EPRs using the reference property `ResourceID` (although any other XML markup could as well be used instead of `ResourceID`). This is illustrated in Figure 6. Note that the endpoint references of I_1 and I_2 contain the same `Address` (`http://abc.com/service`) and only differ in the value of `ResourceID`. In the figure, the SOAP message exchange between the client and the WS-Resources is depicted. WSA specifies that all sub-elements of `wsa:ReferenceProperties` occurring in an EPR are put to the SOAP header. In our example this concerns only the `ResourceID` header. The WSA header `To` holds the value of the `wsa:Address` element of the EPR. As mentioned before, service factories might also operate differently and, e.g., return a new URL for each created service.

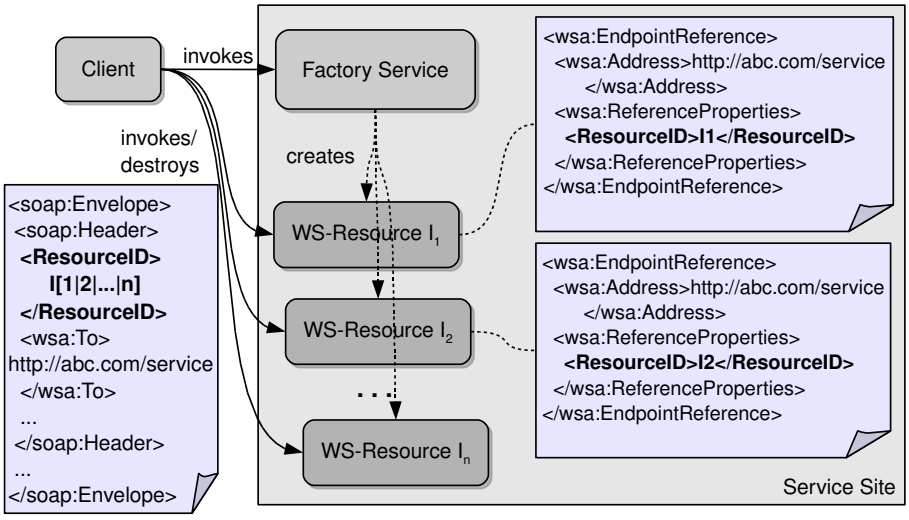


Figure 6: Example of WS-Addressing for Newly Created WS-Resources

2.4 Asynchronous Service Invocation

Synchronous service calls send a request message to a target service and receive a response message with result data as soon as the processing has finished. In the time between request and response message, the underlying network connection remains established. With asynchronous invocations, on the other hand, the request of an operation is separated from the receipt of the response message. According to [97], the four patterns for asynchronous invocations are:

- *Fire And Forget*: A request message is sent and no response message is expected. The sender does not get any acknowledgment of the target service having received the request.
- *Sync With Server*: Similarly to *Fire And Forget*, *Sync With Server* is a one-way request which does not expect a response message. But, in contrast, the receiver acknowledges the receipt of the message on network level.
- *Poll Object*: An operation is requested and the service returns a *Poll Object*. This object is used to either repeatedly query for the result until it is available or to block on the object until the result is available.
- *Result Callback*: After the sender has requested a service operation, the service performs the computation and actively notifies the sender of the completion.

Especially the *Result Callback* version imposes additional requirements on the sender (client) in the form of an interface which enables the service to send its response message. In Web service computing, several possible solutions exist to provide a callback mechanism, three of which are:

- *WS-BaseNotification*: WS-BaseNotification [55] specifies a SOAP-based message format for creation and destruction of *notification subscriptions* as well

as notifications themselves. By sending a `Subscribe` message, clients specify which types of notifications they are interested in and the service answers with a `SubscriptionReference` message. This reference is used to distinguish subscriptions and can be used to poll the service for new messages. When an event occurs that the service wants to inform about, it sends a `Notify` SOAP message to all clients which have subscribed for this type of notification. Listing 4 shows an example `Notify` SOAP message of a `ShoppingService` service which informs about the new status (`SHIPPED`) of an order with ID `order15034`.

```
<soap:Envelope ...>
  ...
  <soap:Body>
    <wsnt:Notify xmlns:wsnt="http://docs.oasis-open.org/wsn/b-2">
      <wsnt:NotificationMessage>
        <wsnt:SubscriptionReference>
          <wsa:Address>http://company.com/ShoppingService</wsa:Address>
          <wsa:ReferenceParameters>
            <NotificationID>order15034-1</NotificationID>
          </wsa:ReferenceParameters>
        </wsnt:SubscriptionReference>
        <wsnt:ProducerReference>
          <wsa:Address>http://company.com/ShoppingService</wsa:Address>
          <wsa:ReferenceParameters>
            <OrderID>order15034</OrderID>
          </wsa:ReferenceParameters>
        </wsnt:ProducerReference>
        <wsnt:Message>
          <OrderStatus orderID="order15034">
            <Status>SHIPPED</Status>
            <Text>Your order has been shipped.</Text>
          </OrderStatus>
        </wsnt:Message>
      </wsnt:NotificationMessage>
    </wsnt:Notify>
  </soap:Body>
</soap:Envelope>
```

Listing 4: Notification SOAP Message

The message contains a `SubscriptionReference` element and a `ProducerReference` element which holds the EPR of the notification producing endpoint. The format of the reference parameter `NotificationID` (the order ID `order15034` as prefix and a trailing `-1`) indicates that more than one notification subscriptions may exist for the same order. The notification body itself is contained in the element `Message`. In the example, the notification informs about the new status of a previously made order. When no more notifications are desired by the client, the subscription is finalized by sending an according `Destroy` SOAP message to the subscription reference EPR.

- *WS-MessageDelivery Callback Pattern*: In WS-MessageDelivery [85], the Callback pattern is used to asynchronously deliver a response message to a request message. The service requester needs to provide a reference to the endpoint which is to receive the callback response (*ultimate response destination*) and a *correlation ID* which is to be returned in the response message later on. Callbacks are declared in the `portType` section of the target service's WSDL doc-

ument: the WSDL `operation` declaration of the request operation is enriched with a `ResponseOperation` element, in which the response operation interface, which has to be provided by the requester in order to properly receive callback messages, is declared.

- *WS-Addressing* `RelatesTo` *header*: In WS-Addressing, the `MessageID` SOAP header element uniquely identifies a SOAP message in the interaction between service consumer and provider. In the reply message sent by the provider, the message identifier of the request message must be present in a `RelatesTo` SOAP header. Therefore, reply messages can be uniquely assigned to request messages and clients can implement parallelism if they perform proper pooling of request and response messages.

2.5 SOAP Fault Handling

Service provider and service consumer in SOAs rely on the exchange of messages over the network in order to to interact with each other. Given the fact that faults and exceptions may occur in this interaction, there is a need for a generalized format for fault messages. In the area of Web services, SOAP addresses this issue by defining *SOAP Faults* [78, 92]. The definition of faults differs between SOAP version 1.1 [78] and version 1.2 [92]. We focus on version 1.2 and briefly discuss the concepts of SOAP faults on the basis of the example given in Listing 5, which contains a fault message indicating that a username/password authentication was unsuccessful.

```
1  <env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope"
2                xmlns:tns="http://www.example.org/authentication"
3                xmlns:xml="http://www.w3.org/XML/1998/namespace">
4   <env:Body>
5    <env:Fault>
6     <env:Code>
7      <env:Value>env:Sender</env:Value>
8      <env:Subcode>
9       <env:Value>tns:InvalidCredentials</env:Value>
10     </env:Subcode>
11    </env:Code>
12    <env:Reason>
13     <env:Text xml:lang="en">Invalid username/password combination</env:Text>
14    </env:Reason>
15    <env:Detail>
16     <tns:Attempts>1</tns:Attempts>
17     <tns:MaxAttempts>5</tns:MaxAttempts>
18    </env:Detail>
19   </env:Fault>
20  </env:Body>
21 </env:Envelope>
```

Listing 5: SOAP Version 1.2 Fault

Each SOAP fault has a `code` and an arbitrary number of `subcodes`. The code gives information on the originator of the fault – possible values are given in Table 2. From SOAP v1.1 to v1.2 the codes `Client` and `Server` have been replaced by `Sender` and `Receiver` and the new code `DataEncodingUnknown` has been introduced. The sub-

Fault Code	Description	v1.1	v1.2
`VersionMismatch`	The version of the SOAP message does not match the expected version.	✓	✓
`MustUnderstand`	A SOAP header with `mustUnderstand` attribute of value `true` could not be processed by the receiver.	✓	✓
`DataEncodingUnknown`	An unknown *encoding* [92] has been specified.	×	✓
`Client/Sender`	The message was incorrectly formed or contained inappropriate information.	✓	✓
`Server/Receiver`	The operation failed due to a failing processing caused by the receiver.	✓	✓

Table 2: SOAP Fault Codes

codes of SOAP faults are used to further distinguish the reason of fault occurrences. Subcodes may be of arbitrary type and are comparable to exception types (exception classes) in ordinary programming languages such as Java. In the example listing the fault subcode `InvalidCredentials` indicates that invalid authentication data has been provided. The `Reason` element in listing 5 gives a human-readable description of the raised fault. The optional `Detail` element provides further details, in our example the number of login attempts made so far and the number of attempts possible (before the service gets locked and becomes unavailable).

2.6 Web Service Composition

In Section 1 we briefly mentioned that intra-service protocol description is essentially a subset of the problem of *service composition*. We will discuss this statement in more detail in this subsection.

The basic Web service infrastructure described in Subsection 2.1 is sufficient to create applications only involving simple interactions between client and server. If the implementation of a service's business logic involves the invocation of other Web services, we speak of a *composite service* [13]. More precisely, in situations where the solution to a problem cannot be implemented with one single, but only with a set of services, some kind of service composition is needed. [13] comprises an analysis and classification of service composition approaches, taking into account *static* and *dynamic*, *automated* and *manual*, *business rule driven*, *model driven*, *context based* and *declarative* composition. In this document we cover only some of these aspects. We discuss WS-BPEL (static, manual, possibly model-driven) and touch on the topic of semantic service composition approaches (automatic, ontology-based, often times

dynamic). The reason why we do not mention model driven, declarative or context based solutions here, is that none of them can be regarded prevalent or state-of-the-art. We will, however, discuss current research in these areas in Section 3.

2.6.1 Service Composition in WS-BPEL

The *Web Services Business Process Execution Language* [56] (WS-BPEL, BPEL) is an XML-based language for the description of electronic business processes, whose individual activities are performed by Web services. The participating Web services are specified by `partnerLink` definitions, which point to the respective `portType` declarations and service EPRs. If the WSDL `portType` definition of a service is regarded as the equivalent of a class interface definition in object-oriented programming (OOP), then the `partnerLink` element in a WS-BPEL process can be seen as an equivalent to the object identifier in OOP: if several instances of a Web service exist, several `partnerLink` definitions may exist with according EPRs. Figure 7 illustrates the mapping of partner links to service EPRs. The figure depicts the two WSDL documents of the process interface and a target service, respectively, as well as an excerpt from a WS-BPEL document and the EPR of one of the target service's instances. Partner link types and roles are defined in the WSDLs and referenced in the `partnerLink` element of the WS-BPEL process definition. The EPR information is assigned to the partner link `service` using an `assign` directive. This EPR is dynamic and may be reassigned in the course of the process execution. The `partnerLink` at-

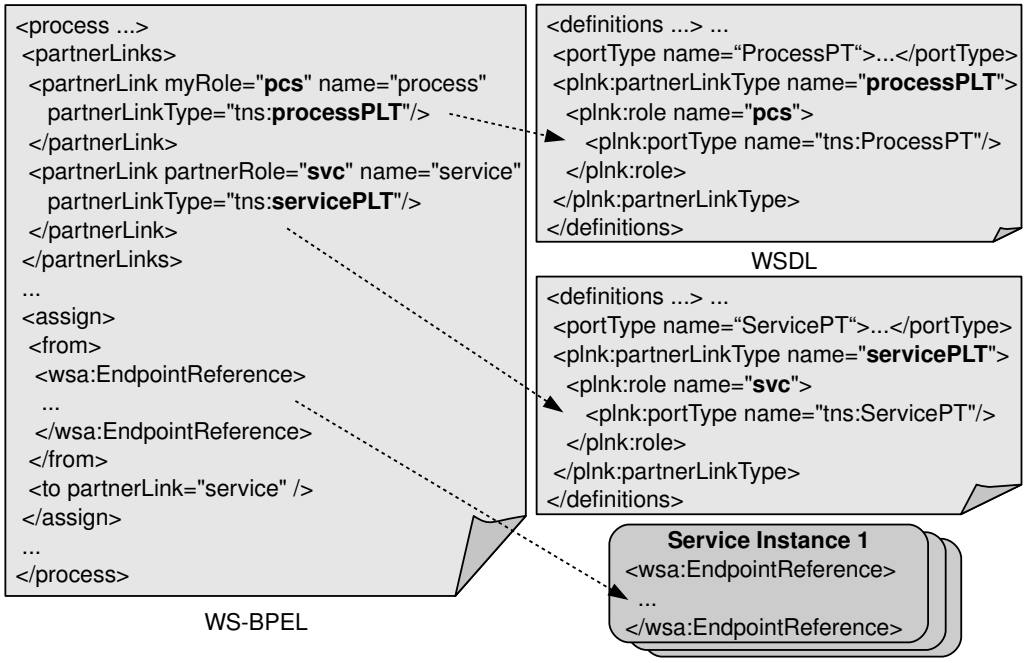


Figure 7: BPEL `partnerLink` Mapping

tribute `myRole` which points to the port type `ProcessPT` indicates that, to the outside world, the WS-BPEL process appears as an invokable Web service with the WSDL

interface defined in this port type.

A WS-BPEL engine parses the process definition, publishes the process WSDL file and accepts incoming SOAP messages which request the execution of the process. Upon receipt of a request message, the engine starts to interpret the XML-encoded directives given in the process definition. The main directives are listed below:

- `invoke`: This instruction invokes an operation of a partner Web service. Input and output of the invocation are defined by one variable each.
- `assign`: WS-BPEL allows for the use of variables which can be assigned values using the `assign` instruction. Values can be atomic (string, integer, ...) or XML markup. XPath expressions are used to select the target part of the variable to be assigned.
- `receive`: The `receive` instruction signifies that a message is to be received from an outside participant. This is used either at the beginning of the process to indicate the start of the execution or in the middle of the process to receive results from asynchronous invocations.
- `while`: repetitive execution of a part of the process.
- `switch`: conditional execution of parts of the process.
- `flow`: parallel execution of several activities.
- `reply`: returns the process result.

BPEL documents are very verbose and usually generated by tools which support graphical development of processes (e.g., Eclipse BPEL Editor [14], Netbeans BPEL Designer [40] or Oracle BPEL Process Manager [51]). Finally it is important to mention that BPEL has been designed for use with more than one target service (see `partnerLink` concept) and is not intended for intra-service protocol specification.

2.6.2 Semantic Web Service Composition

Discovery, execution and composition of Web services with "conventional", well-established technologies (UDDI, SOAP, WS-BPEL) requires, to a large extent, manual work by the human programmer, which limits scalability and curtails the added economic value of Web services [12]. Recent research in the *Semantic Web* community aims at furnishing Web services with more machine-processable semantic markup. The semantic description of Web services enables automatic (and therefore more flexible and cost-effective) discovery and selection of a set of suitable Web service operations, given a characterization of the required functionality. The drawback of providing semantic descriptions is that service creation requires additional expertise and, generally, becomes more time-consuming. Another limitation is that the complexity of service client implementations increases as it becomes necessary to interpret the syntax and semantics of the formal language. But, additional computational complexity can not be dealt with efficiently on all hardware platforms,

e.g., considering mobile devices. Notable specifications in the area of Semantic Web services and Web service ontologies encompass the *Web Service Modeling Ontology* (WSMO) [68], *Ontology Web Language* [83] for Web services (OWL-S) and *Web Service Semantics* (WSDL-S) [88]. A discussion of the concepts of these standards and similarities and differences among them would go far beyond the scope of this thesis. The general idea is to create a formal foundation for the description of a Web service and its "intra" and "inter" relationships in order to create a very exact picture of the service. By introducing inference rules and defining a target functionality (or target state) it becomes possible to dynamically select and compose services suitable to reach the goal.

2.7 Model-Driven Architecture

Model-driven architecture (MDA) (as well as *Model-driven development* - MDD) [43] is a registered trademark of the *Object Management Group* (OMG) for a design approach for the development of software systems. The MDA paradigm is based on platform-independent models, most of which are collectively known under the name *Unified Modeling Language* (UML) diagrams [46]. UML defines several types of diagrams to model different aspects of a software system: 6 types of *structural* diagrams and 7 types of *behavioral* diagrams [44]. According to [3], the underlying motivation for MDD is to improve productivity, in fact, in the short term and in the long term. The short-term productivity depends on how much executable functionality can be generated from the models. The long-term advantage of MDD is in the software artifacts' reduced sensitivity to change. The authors of [33] further state that MDD can help achieve a clear *separation of concerns*. MDD has found its way to various fields of software development, and also to the area of Web service creation and composition. In Section 3 we will discuss current research approaches of MDD in the area of Web service composition and compare it to the proposed solution of model-driven SEPL development, which is explained in more detail in Section 4.

3 Related Work

In this section we discuss work in the field of intra-service protocol description, dynamic service invocation and model-driven service development.

3.1 Web Services Conversation Language

The *Web Services Conversation Language* (WSCL) [80] 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. The main elements of WSCL documents are *conversations*, *interactions* and *transitions*. A conversation can be thought of as the sum of all interactions (message exchanges) with a service in order to achieve a certain functionality or to reach an specific end state. Interactions model the message exchange (or *document* exchange) between two participants. Transitions specify the ordering relationships between interactions. Essentially, WSCL models conversations as finite state machines, in which the states are represented by the interactions. The transitions determine which interactions can be executed after having received a certain output from another interaction. 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. In WSCL the decision, which operations $op_1, op_2, \ldots op_i$ can be executed after an operation op_0 is simply based on the *output type* of op_0. The output type, according to the WSDL contract, may be some XML message m_0 in the normal case or fault messages $m_1, m_2, \ldots m_j$ in exceptional cases. For each message $m_0, m_1, \ldots m_j$ a transition to a subsequent interaction (or an end state) is specified. But, in comparison to SEPL, WSCL allows only for distinctions concerning the type and not the actual content of messages. Finally, WSCL does not define executable protocol functionalities but is rather a guideline for the interaction with a service.

3.2 Web Service Choreography Interface

The *Web Service Choreography Interface* (WSCI) *"describes how Web Service operations [...] can be choreographed in the context of a message exchange [...]"* [75]. The specification describes that WSCI addresses the *dynamic interface* of a Web service and argues that standard Web service description using solely WSDL lacks certain aspects: the WSDL definition does not define how to interpret a set of operations happening in a given order (*choreography*); it is further not clear how to distinguish sub-

sequent operations from different protocol executions or how to associate subsequent operations being part of the same execution, respectively (*correlation*); WSDL definitions do not describe if the Web service operations are performed in a transactional way (*transaction*); real life services may be able to choose different executions based on the informations in the message exchange (*possible choices*). The language constructs of WSCI provide solutions to these issues. Choreography aspects are modeled using the `action` directive and control-flow related instructions (such as `sequence`, `foreach` or `switch`). The correlation mechanism determines for each action which correlation identity it is assigned to. Certain actions, usually `process`-initial actions, instantiate a correlation identity for further use. The `compensate` construct allows for compensation routines in case a sequence of actions cannot be completed successfully in the context of a transaction. Furthermore, WSCI provides methods to manage exceptional situations, such as timeouts and fault messages. WSCI can be seen as a description of the dynamic service interface which benefits the service consumers, who get to know the expected behavior of invocation sequences. Similar to WSCL, WSCI does not define the exact transformation of input and output messages and is therefore – in comparison to SEPL – not directly executable.

3.3 WSDL 2.0 Message Exchange Patterns

In WSDL version 2.0 [89], *message exchange patterns* (MEPs) are introduced as *"a template for the exchange of one or more messages, and their associated faults, between the service and one or more other nodes as described by an Interface Operation component"*. In contrast to WSDL 1.1, operations in WSDL 2.0 can have multiple inputs, outputs, incoming faults and outgoing faults. A MEP defines for a specific operation the type of message exchange that the service performs with one or more partner services. The concepts of MEP yields a change from the procedural way of regarding Web services to a more message oriented way. The WSDL 2.0 specification defines eight predefined MEPs, whereas service providers are not limited to these MEPs, but may define new ones:

- *In-Only*: the service receives a message without returning a response.
- *Robust In-Only*: the service receives a message and returns a fault message if a fault occurs.
- *In-Out*: the service receives a message and returns a response message.
- *In-Optional-Out*: the service receives a message and optionally returns a response message.
- *Out-Only*: the service sends a message (without awaiting a response).
- *Robust Out-Only*: the service sends a message and receives a fault message in case a fault occurs with the partner service.
- *Out-In*: the service sends a message and receives a response message.
- *Out-Optional-In*: the service sends a message and optionally receives a response message.

The authors of [42] perform an analysis of the expressiveness of MEPs by presenting a sample business scenario which involves message exchange among three different types of service. The paper comes to the conclusion that MEPs in WSDL *"can in principle express the communication with different node types and multiple instances of one node type"*, but that MEPs *"are not expressive enough to define the overall interaction of a choreography [...]"*. Furthermore, we argue that MEPs are a step towards capturing the dynamic service interface in WSDL, but that they are not suitable to fully express intra-service protocols of Web services.

3.4 SOAP Service Description Language

The *SOAP Service Description Language* [65, 66] (SSDL) is an effort to create a holistic description of Web services, which focuses on messages and service protocols and which is specifically tailored to integrate with the concepts of the messaging protocol SOAP. The authors argue that WSDL has several shortcomings and has misled to an object-oriented and RPC-style mindset in service computing. Given the fact that SOAP is the most widely used protocol for message exchange between Web services, service descriptions ought to be more SOAP-centric and message definitions should rely on XML instead of creating a new component model simply for contract description. SSDL contracts are XML-based documents which are split up into four major sections:

- *Schemas*: Similar to the WSDL `types` section, the `schemas` section contains XSD definitions.
- *Messages*: The `messages` section defines the format of normal messages as well as fault message documents, which may occur in the message exchange with the service. The message description is SOAP-centric and body as well as header elements can be specified.
- *Protocols*: The `protocols` section provides an extensible mechanism to specify the protocols (i.e., the messaging behavior) that the service supports. The contracts included in this section range from simple MEPs to multi-message interactions and process/workflow definitions.
- *Endpoints*: The `endpoints` section contains WS-Addressing EPRs for the endpoints of the described service.

Especially the `protocols` section is of interest and related to SEPL service protocol descriptions. The core protocol descriptions suggested by SSDL are located at different abstraction levels and address different concerns:

- The *Message Exchange Patterns* framework [63] utilizes the MEPs defined in WSDL 2.0 to characterize the message exchange with partner services.
- The *Communicating Sequential Processes* (CSP) framework [62] lays down the service protocol in a process manner. By means of sequence and choice di-

rectives, service providers describe the expected control flow of multi-message exchanges. In contrast to SEPL, CSP does not describe executable protocols because they lack exact information about data transformations to be carried out. The CSP process definition is rather a guideline for client implementors to stick to the expected behavior. CSP is silent about contextualization of the multi-message interaction and states that standards such as WS-Context [53] or WS-Addressing need to be utilized to clarify which message belongs to which context/conversation.

- The *Rules* framework [64] for SSDL provides a means of expressing relationships between messages in terms of what messages have or have not been sent/received. It imposes constraints on the types of messages which may or may not be sent to and received from the service at a specific point in time.
- The *Sequencing Constraints* [95] framework presents a language to express the valid sequence of (externally visible) actions a service may perform. It is based on the *pi-calculus* [39] and provides constructs for action *sequences, choices, replications, parallel compositions* and atomic *send* and *receive* actions.

The SSDL framework provides the expressive power to describe SOAP-based Web services in a holistic way, i.e., concerning the static as well as the dynamic interface. However, SSDL contracts are rather a guideline for clients and certainly not an executable intra-service protocol. The SSDL `protocols` section is extensible, for which reason we propose to embed SEPL code in SSDL contracts in order to describe the protocol of Web services in an executable way.

3.5 The Web Service Programming Language XL

The authors of [19] 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 [70]). 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* [77] to access certain elements and attributes of XML variables. SEPL additionally supports a "dot-syntax" which resembles the syntax to access class members in object-oriented programming; on the other hand, XL additionally supports *XQuery* [93] statements, which operate on XML data sources and should serve as a replacement for queries to external (relational) databases. In gen-

eral, XP is designed to contain much of the business logic and does not necessarily require an existing target Web service whereas SEPL documents are rather slender and delegate most tasks to the target service implementation. XL and SEPL use a different conversation pattern: 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 uses the service instance EPR to distinguish conversations and, most importantly, publishes its functions as stateless operations which do not require clients to consider any conversation-specific aspects. XL allows for parallel execution which is not yet supported in SEPL. In SEPL it is possible to perform asynchronous invocations using a notification mechanism, which is not directly supported in XL.

3.6 XLANG/s

Microsoft's *XLANG/s* [10] is a service description language for embedded use with .NET-based objects and messages. On the one hand, XLANG/s supports the usage of .NET elements (e.g. C# objects) to create service implementations. On the other hand, XLANG/s provides **call** (synchronous) and **exec** (asynchronous) operations for interaction with partner services which opens possibilities for service orchestration and service protocol specification. XLANG/s is proprietary and neither well documented nor maintained and plays only a subordinate role in the world of Web service standards.

3.7 Dynamic Service Invocation with Daios

Daios [34, 35] is a framework of Java classes which enables dynamic invocation of Web services. The implemented SEPL execution engine is based on Daios, i.e., invocations of Web service operations are realized using the Daios client library. In the following we briefly describe the main features of this framework.

Daios follows a message-oriented approach, i.e., client developers do not "call operations" but send and receive messages to and from the service. Daios uses a special message format - the according class is named `DaiosMessage` - which abstracts messages from their XML representations. The Daios message format is conceptually similar to objects expressed in the data-interchange format JSON [31] (*JavaScript Object Notation*). In JSON, objects are textually represented as a collection of name/value pairs, which is equally the case for Daios messages. The value of one such pair entry may be either 1) a simple type (string, integer, ...), 2) an array of simple types or 3) a message or an array of messages (recursive construction). Listing 6 contains a code excerpt in which a `DaiosMessage` representing a "person object" is created. In lines 3 and 4, the simple values *name* and *age* of the person are set. `DaiosMessage` objects can also be constructed recursively, which is illustrated in lines 5-8.

```
1  DaiosMessage message = new DaiosMessage();
2  DaiosMessage person = new DaiosMessage();
3  person.set(``name'', ``Jack King'');
4  person.set(``age'', 36);
5  DaiosMessage address = new DaiosMessage();
6  address.set(``street'', ``2 King's Cross'');
7  address.set(``city'', ``London'');
8  person.set(``address'', address);
9  message.set(``person'', person);
10 ServiceFrontend f = ... // creation of service frontend omitted
11 f.setWSDLOperationName(new QName(``addPerson''));
12 DaiosMessage result = f.requestResponse(message);
```

Listing 6: Construction of a Daios Message

Once a `DaiosMessage` has been constructed, the class *ServiceFrontend* is used to perform a service invocation. In line 11, the name of the target operation is set. Finally, the message is sent to the target service in line 12 using a synchronous invocation. Beside the "request response" invocation flavor, Daios also supports asynchronous communication ("fire and forget", "poll object", "callback"). The achievement of the Daios framework is that all information necessary to construct the final SOAP message (qualified operation names, XML namespaces, ...) is collected from the service's WSDL document in the background.

The Daios framework features the invocation of WSRF-style services and the use of factory services to create new service resources. We will discuss in Section 4 how these capabilities are utilized in the SEPL framework.

3.8 Model-Based Service Development

MDD has found its way to various fields of software development, and also to the area of Web service creation and composition. In [27], an approach for UML-based composition of Grid services is presented. Based on UML *activity diagrams*, a *domain specific language* (DSL) is defined. In the DSL, each activity represents a Grid service operation. The name of the activity equals the name of the operation. Object flows model the types of messages which are passed from one operation to the other. Tagged values, which are appended to the activities, specify the WSDL location of the target service and whether the service is stateful. In case of a stateful service, the DSL semantics specify that the engine executing the composition shall take care that the same instance is reused for all invocations to this service.

The authors of [24] propose another approach to UML-based service composition that is related to modeling SEPL protocols. The following success criteria are defined:

- *Expressing Web service patterns*: The UML model needs to be expressive enough to support Web service invocations and the basic control patterns (`sequence`, `choice`, `merge` etc.).

- *Readability*: The model shall be easy to understand for experienced modelers.
- *Executability*: It shall be possible that the UML model is transformed to a workflow document (e.g., WS-BPEL), which can be executed by a workflow engine.
- *Independence of workflow language*: The model shall not be tied to one particular target workflow language.

The paper then discusses Web service workflow patterns one by one. *Web service calls* are identified using a triple *(WSDL document, service, operation)*. The *loop* pattern is modeled with a circular control flow path starting with a decision node. In the paper it is mentioned that this approach is a workaround for the UML `LoopNode` which has not yet been defined at the time of writing. The document further discusses the problem of *data transformation* – how the value (parts) of one data object (DO) may become (part of) the value of another DO – and present three solutions: 1) include flow links between the DOs (which is problematic if more than one link points to one DO), 2) provide a mapping function on the flow between activities, 3) introduce mapper actions which are part of the flow and explicitly describe the transformation (e.g. `do1.part1 = do2` for DOs `do1` and `do2`). Depending on the type of transformation (`one-to-one identical/non-identical, many-to-one` the authors propose each one of the solutions. As we will discuss later on, SEPL MDD uses the third approach as the most expressive and powerful one. However, SEPL activity diagrams go without DOs but utilize *input pins* and *output pins* on actions to specify the used variables. The last point discussed in the paper – discussion of *alternative service selection* – is not directly related because SEPL service protocols target only a single service.

In [32], an even broader picture is drawn for different abstraction levels and kinds of models:

- On the **collaboration level**, UML *collaboration diagrams* [45] are chosen for modeling the high-level roles of participants in business interactions. This level is irrelevant for SEPL as intra-service protocols do not target inter-business activities.
- On the **transaction level**, three diagrams model different aspects: 1) *Activity diagrams* describe the behavior of a collaboration in terms of transactional processes. Each action in the diagram specifies the involved participants and the name of the action. The document emphasizes that no central control is required. Rather, each participant must behave in the specified way in order for the resulting behavior of the overall collaboration to correspond to the defined collaboration activity. This is a main difference to SEPL models which are designed to be centrally executed. 2) *Class diagrams* model the structure and attributes of objects used in interactions. With the aid of a detailed data model, the activity diagrams can specify exact constraints and decision guards for data objects. Guards and constraints can equally be defined in SEPL, with the

difference that object types are not static but dynamically resolved at run time. Therefore SEPL models require no information about the class structure of data. 3) *State diagrams* model the abstract states and state transitions which apply to data objects used in transactions. Based on the object states, constraints and decision guards can be defined in the activity diagrams. In SEPL, the state of services is distinguished by means of WS-Resource properties and the state of exchanged message objects is defined by the concrete data contained therein. Hence, there is no need to define abstract states in state diagrams.

- On the **interaction level**, which is a refinement of the transaction level, two diagrams are used: 1) *Sequence diagrams*, in combination with OCL constraints, give a fine-grained description of the actions listed in the activity diagrams on the transactional level. Whereas actions on the transactional level are rather abstract constructs, the interaction level represents explicit and executable sequences of Web service operations. 2) *Class diagrams* are used to model the request and response messages. These messages are based on and use the classes defined on the transactional level. SEPL does not model messages with class diagrams and also does not use a static type system, but dynamically detects message types and extracts type information from WSDL files at runtime.

Although this approach targets the more extensive problem of Web service collaboration protocols, it contains some interesting points related to SEPL. Essentially, by using activity diagrams to model the concrete interaction with a target service, SEPL lifts the interaction one level higher compared to the presented approach. Thereby, SEPL activity diagrams combine the transaction level and the interaction level to a common level which is expressive enough to handle both aspects. This is due to the fact that in intra-service protocols there are only two participants, the client and the target service – plus the special case of an optional factory service.

3.9 Petri Net-Based Web Service Composition

The *Petri net* [67] (PN) is a formal concept widely used to model and simulate the dynamics of computer systems and automata. A PN is a directed bipartite graph, in which the nodes represent *places* and *transitions*. The places contain zero, one or more *tokens* which indicate whether the place is "active". Transitions have the ability to *fire*, which means to move tokens from one state to another. A transition can only fire if all preceding states contain at least one token. If the transition fires, each one token is removed from all preceding states and each one token is added to all succeeding states.

Several research approaches suggest the use of Petri nets to model Web service compositions. [25] uses a Petri net based algebra to provide a formal framework for the composition of Web services. In this algebra, each place represents a (starting, intermediary or final) state in the protocol execution, the transitions denote the invocation

of Web service operations and directed arcs define the control flow of the composition, i.e., the sequence and parallelism of invocations. More formally, the paper defines the term *service net* as a tuple $SN = (P, T, W, i, o, l)$, where P is the set of places, T is the set of transitions representing the service operations, $W \subseteq (P \times T) \bigcup (T \times P)$ is the flow relation, i/o are the input/output places and l is a labeling function which assigns operation names to the transitions. Based on this formulation, the authors define patterns to express typical composition techniques in service nets, mainly *operation sequence, choice, iteration, parallelism* and *service selection*. Figure 8 illustrates these service net patterns. Rectangles with dashed borders stand for any arbitrary subpart of the service composition (which are assumed to start and end with one single place each), places are drawn as circles, transitions are shown as black rectangles, and black dots inside the circles depict the available tokens in a place. Part a) of the figure shows the pattern for the operation sequence; composition parts $S1$ and $S2$ are simply connected with a separating transition. A choice can be modeled by prepending each a transition to the subparts $S1$ and $S2$ as well as a place with one token in front of the transitions (see part b)). Clearly, only one of the transitions can

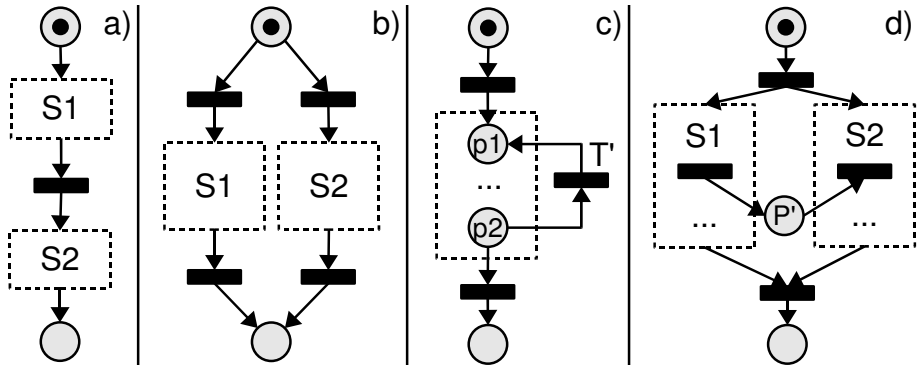


Figure 8: Petri Net Service Composition Patterns

fire, because only one token is available in the place and hence the pattern produces an exclusive choice. Part c) depicts the iteration pattern where a loopback connection (via a transition T') is inserted between the start place $p1$ and the end place $p2$ of a composition subpart. In the control flow at $p2$, either T' is fired to start a new iteration, or the final transition is fired to leave the iteration. In part d), a simplified picture of the parallelism pattern is given. Upon firing of the transition which precedes subparts $S1$ and $S2$, both subparts become active and can execute in parallel. To achieve communication and synchronization between $S1$ and $S2$, a place P' has been inserted which is connected to each one transition in $S1$ and $S2$, respectively.

A similar Petri net based approach, called *WS-Net*, has been introduced in [98]. WS-Net describes a Web services oriented software architecture as a set of connected architectural components and distinguishes between three levels:

- *Interface Nets* define the interface of architectural components as a set of semantically related operations which the component provides.

- The *interconnection net* specifies which foreign service operations are required in order for a Web service to operate properly.
- The *interoperation net* describes the connections between service components and the data flow for the entire system.

The advantage of the Petri net model lies in the support for many flow concepts (e.g., choice, parallelism, iteration) and its formal mathematical 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. UML activity diagrams, which are generally more light-weight than Petri nets, can be transformed to a Petri net representation as shown in [69]. This transformation approach can be used to generate Petri service nets from SEPL activity diagrams in order to apply correctness checks on the service nets.

4 Design

In this section we present the architectural design of the SEPL framework - the intra-service protocol modeling, description and execution framework. After an introductory example scenario description, the design section is split up into 3 main parts: firstly, we define the syntax and semantics of the service protocol language SEPL; secondly, the service protocol modeling features are presented along with a mapping of SEPL features to UML activity diagram elements; thirdly, we describe the architecture of the SEPL service protocol server. In this section we focus on the conceptual ideas of the framework, whereas details concerning the prototype implementation can be found in Section 5.

4.1 Example Scenario

As the motivating example of a service protocol we consider an imaginative service hosted by a European cell phone operator (CPO). The service allows other competitor 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

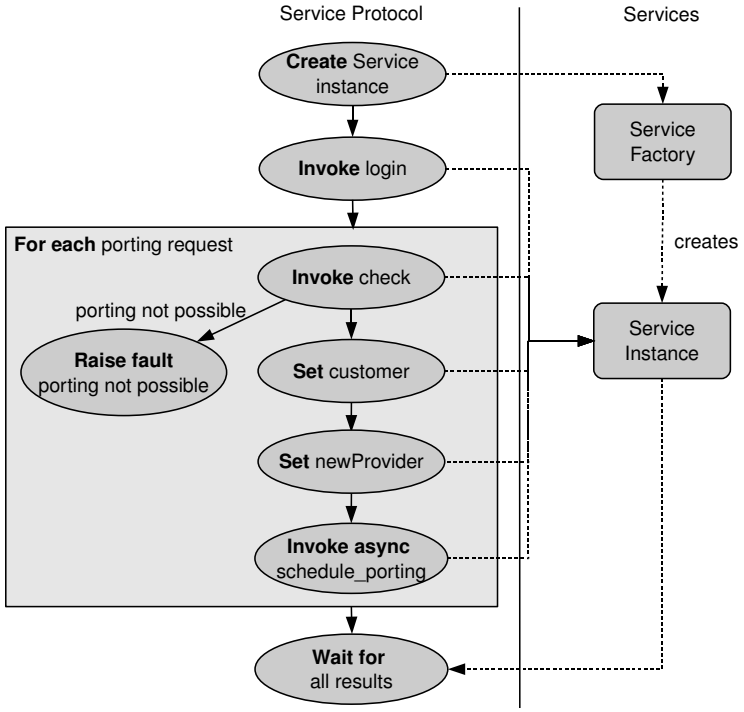


Figure 9: Porting Service Protocol

factory/instance pattern (one stateless factory service is used to create stateful Web service resources). Figure 9 sketches the protocol of the service in a simple graphical notation. On the left-hand side of the figure the actual steps that have to be

carried out by the client are sketched, while the right-hand side shows the services involved. Before being able to access any functionality for number porting the client has to create a new service resource, which handles the request. It does so by using the `Service Factory`'s `create` operation. The `Service Factory` then creates a new stateful service resource for the client, and returns a reference to this instance. The first step that the client then has to carry out at the new service resource is to log in, in order to be granted access to the actual service functionality. The protocol now loops over all porting requests which each contain information on the customer account to port and the new provider to port the number to. In the loop, it should first be checked whether the porting process is currently possible. If it is not, the client has to escalate the issue (i.e., by raising an application exception, interrupting the current execution). 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, which will cause 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 notifications have arrived and the results of all porting operations have been collected.

Even though the protocol of this service 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 [82], and handling of WSRF resource properties. We consider these challenges as "standard" for intra-service protocols, meaning that many real-life stateful Web services utilize concepts such as the ones mentioned. Note that the problem of formally defining this protocol is similar to Web service composition, however, the services involved are limited to the actual Web service that the protocol considers, and, as a special case, the factory service that is used to create new resource instances for this particular service.

4.2 SEPL - The Service Protocol Language

The key design principles of the high-level service protocol specification language SEPL are the usage of an intuitive syntax and practicability of Web service technology. Hence, SEPL combines syntax concepts taken from popular programming/scripting languages (e.g., Java or JavaScript) and satisfies a set of relevant Web service specifications including WS-Addressing, WS-ResourceProperties and WS-Base-Notification. In the following subsections we first present a reference implementation of the example scenario in SEPL, on the basis of which we then define the syntax and semantics of SEPL.A complete syntax description of the SEPL language in *Extended Backus-Naur Form* [29] (EBNF) is given in Listing 17 in Appendix B.

4.2.1 SEPL Example Protocol

Listing 7 shows the implementation of the illustrative example from Subsection 4.1 in SEPL syntax.

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

Listing 7: Number Porting Service Protocol in SEPL

The *function* `port_numbers`, which embraces the SEPL code, defines the input parameters needed by the service protocol. Inside the function, `factory.createResource()` indicates the start of a new porting activity. Afterwards the protocol as described in Subsection 4.1 is mapped to invocations of the newly created service resource and interactions with WS-Resource properties. Finally, the porting results are received asynchronously and returned to the client executing the protocol, after which the resource is destroyed again. For the sake of completeness, the protocol also handles various error conditions that can happen during the protocol execution. The details of the SEPL instructions used in this example are explained step by step in the subsections to follow. Line numbers in the following sections always refer back to this listing.

4.2.2 SEPL Basics

The top-level structure in SEPL documents is called **function**. Each function has a name and a list of parameters, and contains one or more instructions. Parameters are *final*, i.e., they must not be assigned values anywhere in the function. SEPL allows for the use of **variables**, which are untyped, i.e., their content is not limited to a specific type. The type and its supported operations are determined by the SEPL interpreter at runtime. Variables are declared when they are first used, so there is no need for

explicit variable declaration. Variables can hold basic types (e.g., numbers, strings), arrays, object references and XML structures, which are treated specially in SEPL (see below). The statement `callbacks[] = ...` used in line 16 is a convenience operation to append an element to the end of an array. If the array has not been declared before, it is automatically created. Some variable names are reserved and must not be assigned values (`properties`, `factory`, `fault`, `async` and `self`). The semantics of these reserved variables are further explained below. In the example implementation in Listing 7 we use a number of variables, e.g., `status` in lines 11-13 to indicate whether porting is currently possible for this customer.

The **invocation of a Web service operation** in SEPL resembles a method call in an ordinary programming language. An invocation contains the name of the WSDL operation as well as a list of parameters, and will return a result which can be directly assigned to a variable. In our example protocol we use 3 service invocations - `login`, `check_porting_status` and `schedule_ porting_for` (see lines 6, 11 and 17 in Listing 7). The name of a service invocation in SEPL is directly mapped to the name of the according WSDL operation of the target service. Figure 10 depicts the relationship between SEPL code, WSDL and SOAP. From the SEPL protocol description we determine the name of the operation and look up its definition in the WSDL to finally construct a SOAP message and send it to the target Web service.

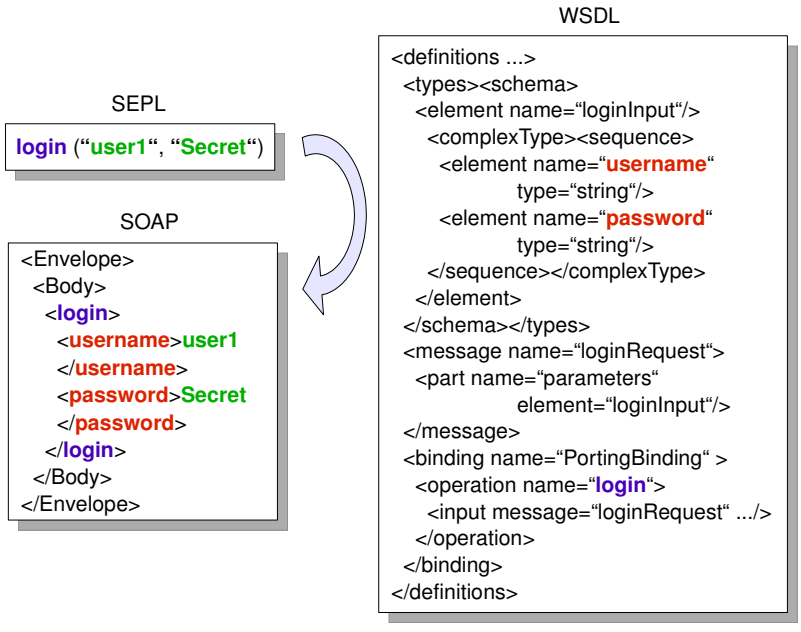


Figure 10: SEPL-to-SOAP Mapping

SEPL allows for the use of **functions** to aggregate multiple SEPL commands. From a client's viewpoint, a SEPL function can be seen as a separate Web service operation on top of the operations defined in the WSDL. Essentially, a SEPL function breaks down a service protocol to an interface contract in the form of the function signature. This is similar to WS-BPEL, where a service composition can again be published as a Web service using a WSDL interface. In our example, the implementation of the

number porting protocol is accessible via a function named `port_numbers` (declared in line 3). SEPL protocol definitions may contain more than one function, and functions can be invoked from within other functions. This increases reuseability and can help avoid duplicate code.

To define the behavior of service protocols, SEPL supports a number of standard **control flow structures** (`if-then-else`, `while`, `foreach`) with semantics akin to languages such as C or Java. SEPL 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. Lines 10-18 of the porting service protocol contain a `foreach` loop which loops over all elements of the array variable `requests`. We use a conditional statement (`if`) in line 12 to check whether the previous invocation of the `check_porting_status` operation indicated that the client may continue. Protocols can **return results** (e.g., indicating outputs of the protocol, or returning status codes) using the `return` statement (see line 22 in Listing 7).

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`. Depending on the content of the variable `var`, the type of `var.element` might be an XML structure or a simple type (e.g., integer, string, boolean). In our example protocol we make use of this approach when accessing the elements `isPossible` (boolean) and `problemDetail` (string) in the response of the operation `check_porting_status` (lines 12 and 13). Listing 8 shows an extended example of how XML elements and attributes can be accessed in SEPL. A string containing XML markup is assigned to a variable `a`. The XML sub-element `<b>` is accessible using the expression `a.b`, and further sub-elements `<c>` using `a.b.c`.

```
1  a = "<a xmlns:ns=\"http://...\">
2          <ns:b>
3            <c>text1</c>
4            <c name=\"name1\">text2</c>
5          </ns:b>
6        </a>"
7
8  a.b.c[0]                    // returns "text1"
9  a.b.c[1].attr("name")       // returns "name1"
10 a.b.c                       // returns {"text1", "text2"}
11 a.b                         // returns object of type 'XML element'
12 a.xpath("b/c[1]/text()")    // returns "text1"
13 a.xpath("b/c[2]/@name")     // returns "name1"
14 a.xpath("b/c")              // returns {"text1", "text2"}
15 a.xpath("b")                // returns object of type 'XML element'
```

Listing 8: XML Usage in SEPL

If an expression like this returns more than one element, these elements are treated as an array. XML attributes can be read and set using the operation named `attr`. The content of simple type elements is directly evaluated: taking the example listing, the

expression `a.b.c[1] * 2` would evaluate to 246. This convenience syntax is equal to XPath with the difference that a dot (`.`) is used instead of a slash (`/`) to access sub-elements. XPath expressions may as well be used directly as can be seen in the lines 12-15 of Listing 8. Note that in XPath array indices start at position `1` whereas in SEPL array indices start with `0`.

4.2.3 WSRF Specific Features

The WSRF set of specifications [58] defines a message exchange model and related XML definitions to access stateful (computational) resources, which retain a state between several 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 state of a resource in WSRF is defined by a set of properties (WS-ResourceProperties). The predefined SEPL variable named `properties` allows **direct access to** such **resource properties**. A property named `prop1` can be accessed in SEPL using the statement `properties.prop1`. In our number porting service in Listing 7 we use resource properties named `customer` and `newProvider` to specify which customer account shall be ported to which new provider (lines 14 and 15). Figure 11

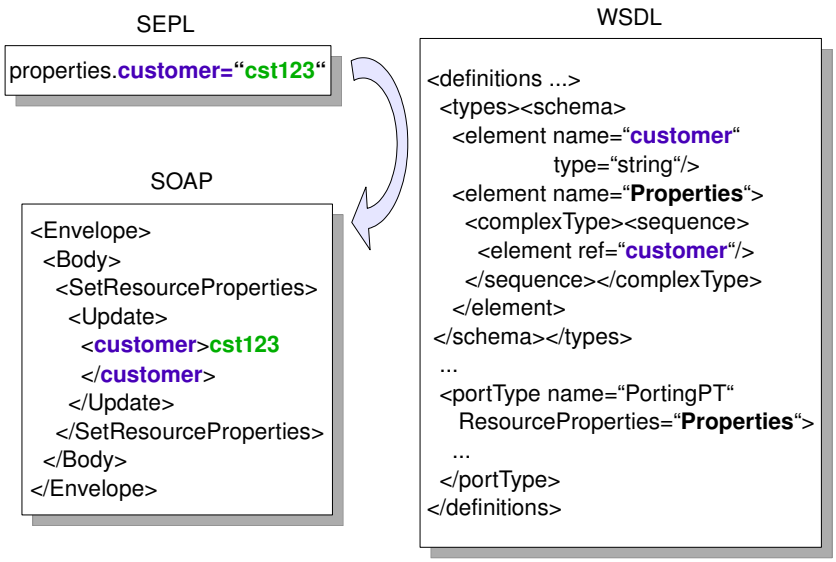


Figure 11: Mapping Resource Properties

depicts the relationship of SEPL's syntax for resource properties to WSDL and SOAP. The WSDL document of the service, through which the resource is made available, contains a definition of the property that is used to verify that the new value has valid content. Then, the SEPL command is transformed to a service invocation with a `SetResourceProperties` SOAP body element.

The WSRF specification suggests the usage of **factory services** to control the lifetime of service resources (often referred to as the factory/instance pattern). The factory service should provide a `create` operation to bring new resources into existence. The response of this operation should contain the WS-Addressing endpoint reference (EPR) of the new resource. In our example protocol, we use a factory service to create a new service resource for each number porting activity. For this purpose, SEPL provides a predefined variable named `factory`. In the SEPL protocol in Listing 7, firstly the address of the factory service is set by assigning the address URL to the variable `factory.EPR` (line 1). This is a convenience operation and, if additional information is needed to address the factory service, the variable `factory.EPR` can also be assigned an XML string containing the markup of a WS-Adressing `EnpointReference` element. Secondly, the factory service's operation `createResource` is invoked to create the new resource (line 4). The response of the invocation contains the new EPR, which gets assigned to the predefined variable `self.EPR`. From this time on, all further service invocations reach the service resource available under the EPR stored in `self.EPR`. Finally, we call the service instance's `Destroy` operation to destroy the newly created resource (line 21) [54]. No further input from the service provider is necessary on how to handle resource creation or termination.

4.2.4 Advanced Concepts

Besides synchronous service operations, SEPL supports the notion of **asynchronous invocations** using WS-BaseNotification. In WS-BaseNotification, consuming Web service endpoints subscribe at the producing endpoint to receive notification messages of a special topic or with a certain message content. The message content that consumers are interested in can be specified using an XPath expression, which must evaluate to a non-empty node set or the boolean value `true` when matched against the body of a SOAP notification message. In SEPL, the predefined variable `async` is used to handle notification registrations and events. In the example in Listing 7, the instruction `callbacks[] = async.WSN( "//PortingResult[@customer='...']")` (see line 16) registers a subscription for notifications at the number porting service and appends the returned callback object to the end of the array `callbacks`. Within this subscription, the callback object will receive all notifications containing an XML element with tag name `PortingResult` and an attribute `customer` with the respective customer identifier. The example is relatively simple, however, XPath allows for queries of arbitrary complexity to filter notifications. After the subscription for a notification, code execution continues until the `wait` operation is called on the callback object. `wait` causes the script to block until an according notification is received from the service. Optionally, a timeout interval can be specified for the `wait` operation to avoid scripts blocking forever when no notification arrives. In our example we collectively `wait` for all results at the end of the protocol function (lines 19-20).

SOAP Faults [78], 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 a brief as well as a detailed description of the fault's reason and its origin. SEPL provides means to handle SOAP Faults using a `try-catch-finally` notation similar to the exception handling syntax in traditional programming languages. Fault handling always happens for a particular fault code, although it is possible to use a wildcard symbol (`*`) to catch faults of any type. Inside a `catch`-block the predefined variable `fault` can be used to obtain details about the fault. Additionally, SOAP Faults can be thrown from inside the protocol using the `throw` keyword. In Listing 7 a fault is thrown in line 13. In the listing, fault handling is performed for the invocation of the operation `login` (lines 5-9). The `catch` block beginning in line 7 catches faults of type `invalidCredentials`. To define a universal `catch` block which catches SOAP Faults of any type, the wildcard symbol `*` can be specified; inside the catch block the predefined variable `fault` is used to access the `Body` element of the SOAP message containing the raised Fault. In the example, we access and return the child element `detail` of the `fault` and return it as the result of the function. SEPL allows for the use of a `finally` block (not included in the example), which is executed at any rate, whether or not any fault has occurred in the execution of the `try` block.

4.3 Model-Driven SEPL Development

Regarding the syntax of SEPL, we see clearly that it requires a certain amount of programming skills to compose service protocols. We therefore present a more convenient way of creating service protocols graphically. The suggested method is based on UML activity diagrams (in the following referred to as *service protocol activity diagram*, or short *AD*).

The main entity of ADs is the *activity*. Just as a SEPL document may contain several functions, an AD may contain several activities. We define that an activity is equivalent to a function in SEPL code. The UML standard defines that an activity contains *executable nodes* and *control nodes* as well as edges between these nodes. The node types are further divided into subtypes as depicted in Figure 12. Executable nodes are logically divided into atomic *action nodes* and *structured activity nodes*. An action node represents a single instruction whereas a structured activity node is a node which internally contains one or more nodes (which may themselves be simple or composite). An action node may contain *input pins* and *output pins*, which determine the input parameters and return value of the action, in case the action is a *call operation action*. A specialization of structured activity nodes are *loop nodes*. ADs use the control nodes *final node*, *decision node* and *merge node*. Final nodes signalize the end of a subflow or of a whole activity. On decision nodes, the control flow is split up into two or more conditional branches. These branches can run together on a merge node. After a merge node, the control flow continues on a single path. Additionally to these three types of control nodes, UML defines the *fork node* and the *join node* [44] which can both be used to control the flow of concurrent

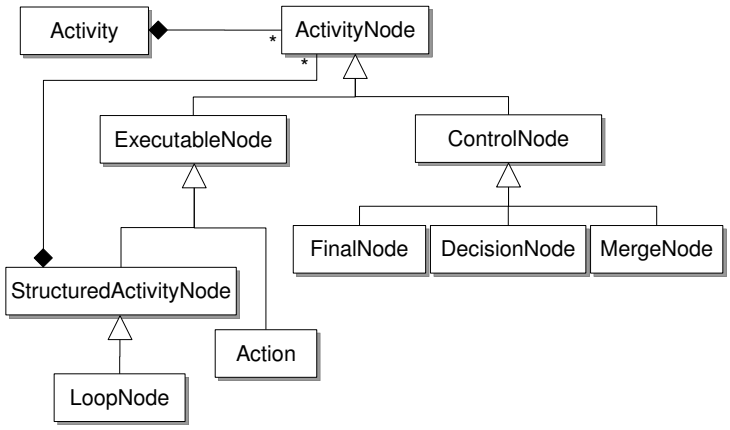


Figure 12: UML Activity Nodes Used in SEPL ADs

computations. But since SEPL does not support concurrency, we will not consider fork and join nodes.

Edges in SEPL ADs are directed and each edge connects exactly two nodes. The directions of the edges signify the control flow of the service protocol. UML distinguishes between *control flow* edges and *object flow* edges. We will not take into account the latter and assume that all edges in SEPL ADs are control flow edges.

We map the mentioned AD nodes and edges to SEPL language constructs (such as service invocations, assignments, loops, branches etc.). For this purpose, we make a distinction between *atomic*, *structured* and *composite* SEPL language constructs. Atomic language constructs consist of a single operation and can be mapped to a single action node. Structured language constructs potentially contain several other operations or language constructs and can be mapped to a structured activity node. Composite language constructs are mapped to a composition of nodes and edges. The difference to structured constructs is that composite structures are made up of several nodes but do not represent a single (structured) node in the diagram.

For a better understanding it is helpful to recapitulate the different language constructs of SEPL. Table 3 contains a listing of all (classes of) SEPL language constructs, including a code example and whether they are atomic or structured. All atomic language constructs, namely service invocations, variable assignments and return statements, can be expressed in ADs using a single action node. Structured language constructs, i.e. branches, loops, function and try-catch blocks, may consist of several nodes and edges in ADs.

Before going into detail with the definition of mappings between SEPL code constructs and AD elements, we consider the following general syntax rules for SEPL ADs:

- Every decision node has exactly one incoming edge and two or more outgoing edges.

Language Construct	Example	Atomic/Structured
Service invocation	login(username,password)	atomic
Invocation/assignment	result = login(username,password)	atomic
Return statement	return result	atomic
Set resource property	properties.prop1 = value	atomic
Subscribe for notification	callback = async.WSN("//Op1Result")	atomic
Receive notification	result = callback.wait()	atomic
If-branch	if(var1 < 10){ ... }	composite
If-else-branch	if(...){ ... } else { ... }	composite
If-else if-else-branch	if(...){ ... } else if(...){ ... } else { ... }	composite
While-loop	while(var1 < 10){ ... }	structured
Try-catch-finally block	try { ... } catch(...) { ... } finally { ... }	structured
Function	function f1(param1,param2){ ... }	structured

Table 3: List of SEPL Language Constructs

- Every merge node has exactly one outgoing edge and two or more incoming edges.
- Every structured node (function, loop, block) contains exactly one node containing no incoming edges OR one initial node (OR no child nodes at all).
- The AD graph must not contain any cycles.

These rules are important to stick to because they determine whether the framework can successfully parse ADs to generate SEPL code. We will discuss this in more detail in Section 5.

4.3.1 SEPL-to-UML Mapping

In the following we define the mapping of SEPL language constructs to UML AD elements (please refer to Table 4 for concrete examples of each of the discussed mappings). Note that the mapping adheres to the UML standard for activity diagrams, and as SEPL ADs require only a small subset of the latter, the mapping contains a few simplifications where reasonable. Specifically, SEPL ADs contain no *object flow* edges but only *control flow* edges and make extensive use of the generic `name` attribute rather than using specialized element attributes. The latter has the advantage that the `name` attribute is visually available on many UML diagram editors and in general simplifies diagram creation.

A **service invocation** in SEPL code is represented by a UML `CallAction` entity. The name of this entity equals the name of the service operation to be invoked. This entity is associated with an ordered set of UML `InputPin`s (zero, one or more `InputPin`s are allowed). The names of the `InputPin`s equal the names of the invocation parameters in the SEPL code. Graphically, the entity is represented by a rectangle with rounded corners containing the name of the operation. Input pins are drawn as rectangles and usually overlap the area of the the `CallAction` rectangle.

An **invocation/assignment** is represented in the same way as a service invocation with the extension that the `CallAction` entity additionally contains one output pin whose name equals the variable to be assigned the result of the invocation.

To model a **return statement** in SEPL ADs, a UML `CallBehaviorAction` entity is used. The name of the entity equals the String `return <statement>` where `<statement>` is the statement (a variable, in general) to be returned. The graphical representation is a rectangle with rounded corners.

To model the **assignment** of a **resource property**, a `CallBehaviorAction` is used, whose name equals `set <property name>`. The action contains a single input pin which holds the value to be assigned.

Notification subscriptions are modeled using a `CallBehaviorAction` with the name `async "<XPath>"`. `<XPath>` is the XPath expression used as the filter to select matching notification messages. An output pin defines the variable to which the callback object gets assigned. Relatedly, the **receipt of notifications** is modeled with the same node type and the name `async wait`. The input pin to this action node holds the name of the callback variable. An (optional) output pin defines which variable the notification message content gets assigned to.

The model of an **if-branch** is a compound structure made up of a decision node, a merge node and an executable node. The decision node has one incoming edge and two outgoing edges, one of which is connected to the merge node. The other edge is connected to the executable node and holds a guard [44] with the condition of the `if`-branch. The executable node represents the body of the if-branch.

The model of an **if-else-branch** is a compound structure made up of a decision node, a merge node and two executable nodes. The decision node has two outgoing edges. One of these edges holds a guard with the condition of the `if`-branch and is connected to the first executable node, which represents the body of the `if`-branch. The other edge is connected to the second executable node, which represents the body of the `else`-branch. Both executable nodes are connected to the merge node.

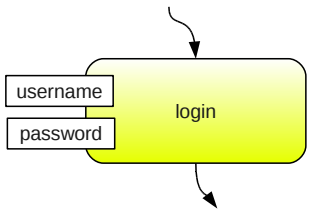
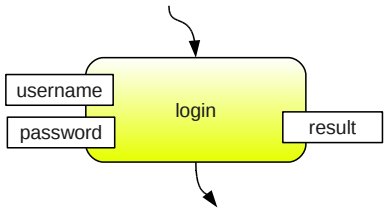
The model of an **if-else if-else-branch** is a compound structure made up of a decision node, a merge node and three or more executable nodes. One of the executable nodes represents the `if`-branch, one represents the `else`-branch and the remaining executable nodes represent `else if`-branches. The decision node has an outgoing edge to each of the executable nodes. The edges pointing to the `if`-branch and the `else if`-branches hold guards with the respective conditions. The edge which connects to the *else*-branch has no guard defined. All executable nodes are connected to the merge node.

The equivalent to a **while-loop** is a UML `LoopNode`. The `LoopNode` is a structured activity node which internally contains one or more nodes. The internal nodes determine the body of the while-loop (specified via the UML association `bodyPart`). The

loop condition is specified in the form of a `decider`, which is an output pin whose value is examined before every loop iteration. If and only if the value evaluates to *true*, the body of the loop is executed. On that account, exactly one of the internal nodes needs to be an activity node with zero incoming edges (this activity node is executed first, followed by all nodes connected to it).

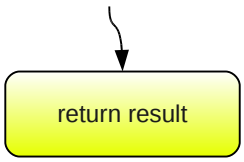
Try-catch-finally blocks are expressed in UML utilizing the `ExceptionHandler` construct. An exception handler can be thought of as a link between two executable nodes (either simple action nodes or structured activity nodes). The source of the link is the protected node (UML association **protectedNode**) and the destination of the link is the handler body (UML association **handlerBody**). Semantically, the protected node makes up a `try` block and the handler body represents either a `catch` block or a `finally` block. The UML association `exceptionType` specifies which type of exception is to be caught: If `exceptionType` is empty, the target handler body is a `finally` block; if `exceptionType` equals *, the handler body is a `catch` block which handles SOAP faults of any kind; otherwise the target handler body is a `catch` block handling SOAP faults whose fault code equals the string value of `exceptionType`. For the reason that any fault being caught in a SEPL `catch` block is always accessed using the reserved variable `fault`, there is no need for the UML association `exceptionInput` and it is generally ignored.

A SEPL **function** is modeled as a UML `Activity`, the top-level entity of UML activity diagrams. The name of the `Activity` maps to the name of the function. The input parameters are specified using `ActivityParameterNodes`. For simplicity, we use the attribute `name` of `ActivityParameterNode` and not its association `parameter`. The order of the input parameters has to be consistent with the order in which they occur in the SEPL function signature. Activity parameters contain no type information. Activities define no return type or output pins, the value returned by an activity is determined solely by the `return` actions occurring therein.

SEPL Code	**SEPL Activity Diagram**
Service Invocation `login(username,password)`	
Invocation/Assignment `result = login(username,password)`	

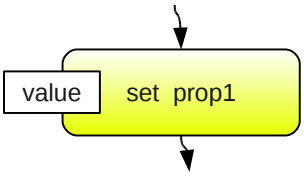
Return Statement

```
return result
```



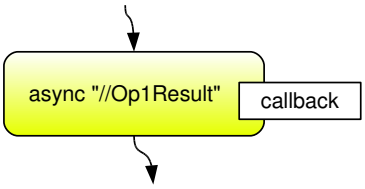
Set Resource Property

```
properties.prop1 = value
```



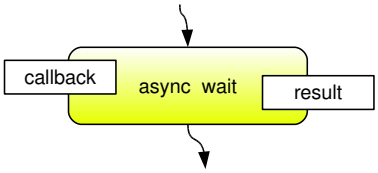
Subscribe for Notification

```
callback = async.WSN("//Op1Result")
```



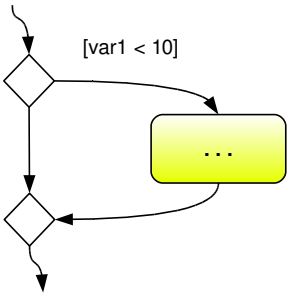
Receive Notification

```
result = callback.wait()
```



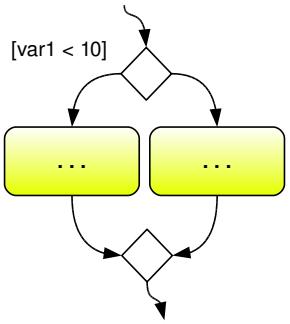
If-Branch

```
if(var1 < 10){
  ...
}
```



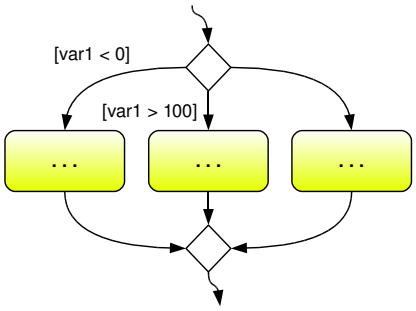
If-else-Branch

```
if(var1 < 10){
  ...
} else {
  ...
}
```



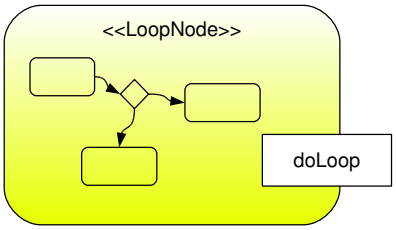
If-else if-else-Branch

```
if(var1 < 0){
  ...
} else if(var > 100) {
  ...
} else {
  ...
}
```



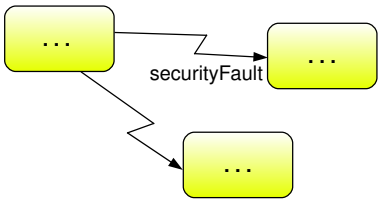
While-Loop

```
while(doLoop){
  ...
}
```



Try-catch-finally Block

```
try {
  ...
} catch(securityFault) {
  ...
} finally {
  ...
}
```



Function

```
function func1(param1, param2){
  ...
}
```

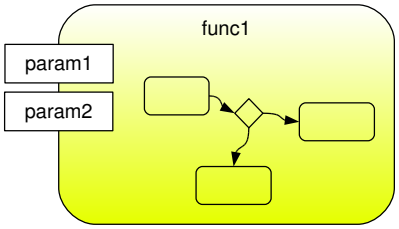


Table 4: Mapping of SEPL Constructs to Activity Diagram Elements

Figure 13 depicts the screenshot of the UML model of the number porting service presented in Section 4. The screenshot has been taken from the graphical presentation of the diagram created in the Eclipse modeling tool *MDT* [15] (see subsection 5.2). The outmost rounded rectangle is the activity `port_numbers` with the parameters `username, password` and `requests`. Inside the activity, actions are drawn in smaller rounded rectangles. From the `login` action, one edge points to the next "regular" node (the first `foreach` block) and one edge depicts an `ExceptionHandler` link. The rectangles with dashed border depict the two `foreach` loops which occur in the protocol: the input pins `requests` and `callbacks`, respectively, denominate the array over which the loop iterates; the output pins `r` and `c`, respectively, denominate the local variable used in each iteration. The action containing `async "..."` stands for the subscription of the notification and the string `$1` inside the XPath query refers to

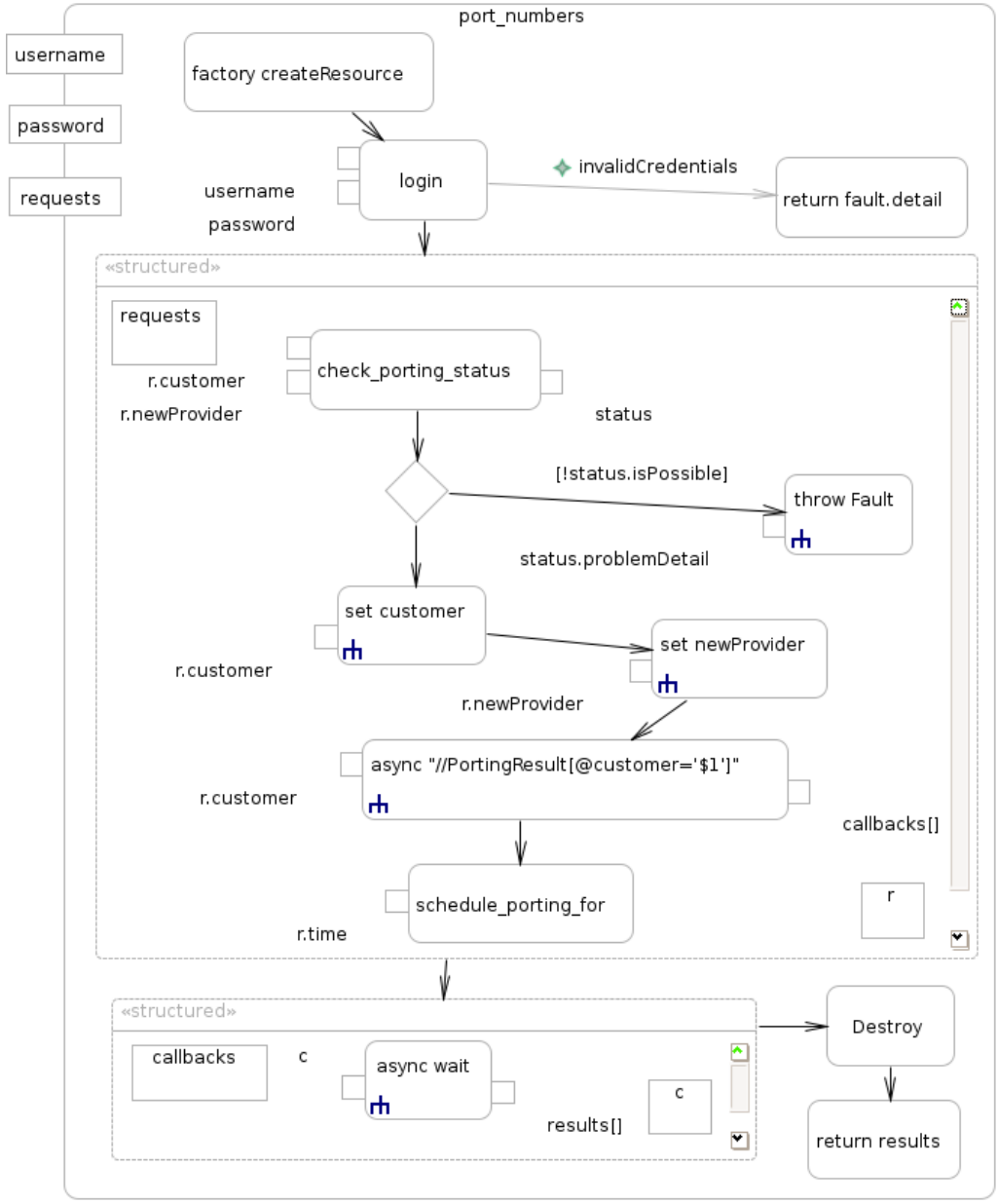


Figure 13: Number Porting Protocol Activity Diagram

the first parameter of this action (`r.customer`) [2]. At the end of the control flow, the figure contains a `Destroy` action and a `return` statement, both depicted in rounded rectangles.

4.4 SEPL Protocol Host

The SEPL client implementation allows for the client-side execution of service protocols. Client-side protocol execution has a number of drawbacks. For one thing,

[2] `$2, $3, ..., $9` can furthermore be used to refer to the second to ninth parameter.

asynchronous communication with services requires the client to open a separate port to listen for notification messages. Secondly, clients require the SEPL client library, additionally to the Web service libraries. We therefore provide for a solution where clients can launch a protocol execution with standard Web service tools. 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).

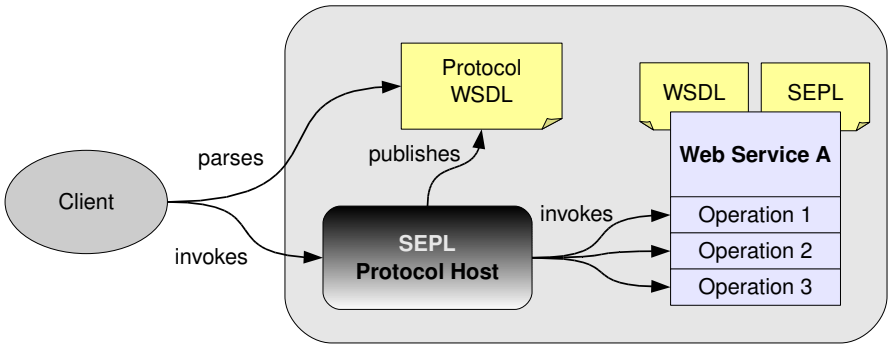


Figure 14: SEPL Protocol Host

Figure 14 depicts a protocol hosting scenario. The PH is configured to host the protocol for a Web service A. The PH has knowledge of the interface (WSDL) and the protocol specification (SEPL) of service A. With the combined information of these two documents the PH generates and publishes a the *protocol WSDL* document, which contains all functions of the SEPL document as WSDL operations. The client - using any standard Web service library - parses the protocol WSDL and sends an invocation message to the PH. The PH receives the request and dispatches it.

We identify three main tasks performed by the PH: 1) generating the protocol WSDL document from the service's SEPL and WSDL documents; 2) dispatching incoming requests; 3) executing the protocol and returning the result. These three tasks will be briefly discussed in the following.

4.4.1 Generating Protocol WSDL Documents

In the protocol WSDL document the data from both the target service's WSDL file and the SEPL protocol is combined to provide a service interface definition covering all protocol functions. The key issue is that SEPL function parameters are untyped and that WSDL, based on XSD, requires type information of operation parameters. Although XSD allows for the use of the special element type `any` for untyped elements, Web service clients can usually handle typed messages better. Therefore we try to extract the maximum level of type information from SEPL documents and avoid the usage of the `any` type wherever possible.

Figure 15 contains example code excerpts from a SEPL document with one function `purchase`, the WSDL document of the target service and the protocol WSDL definition which is generated from this information. Three service invocations occur in the SEPL function `purchase` (`login`, `addToCart` and `submit`). The binding style of the WSDL is document/literal [94] *wrapped*, i.e., in the XSD the operation parameter elements 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.

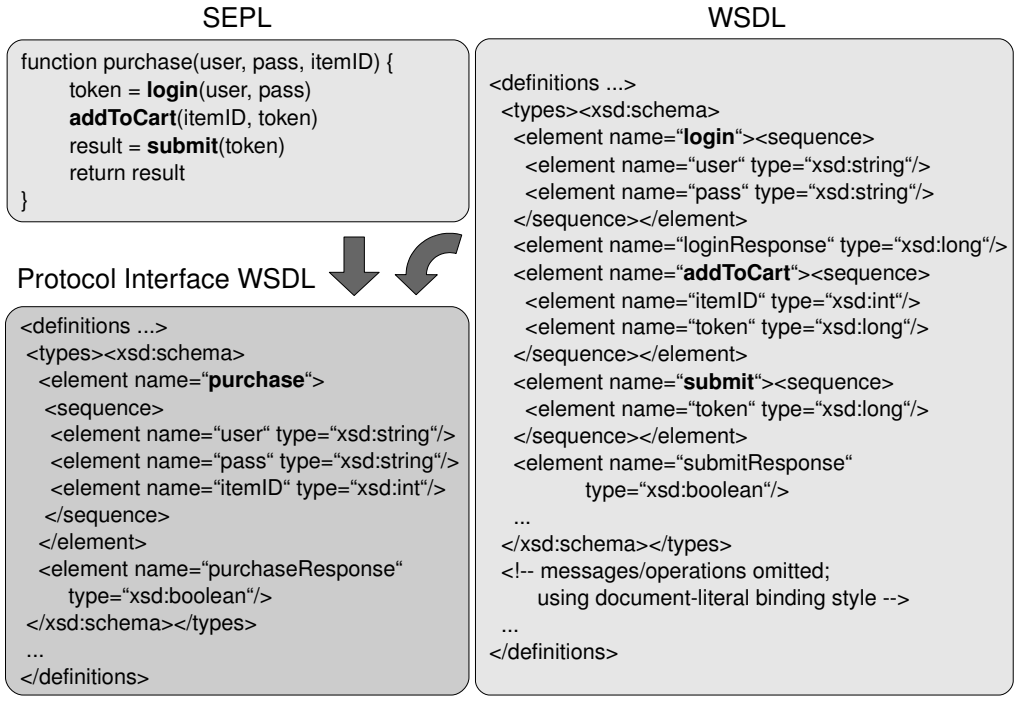


Figure 15: Generated Protocol WSDL Document

For the WSDL generation, with regards to the example in hand, we consider the following:

- The name of the XSD top element (wrapper element) equals the name of the protocol function (`purchase`).
- 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 WSDL's `types` section), hence the type of the function's parameters `user` and `pass` are assumed to be of type `string`.
- The variable `itemID` is passed to the invocation of the operation `addToCart` as the first parameter. We read out from the WSDL `types` section that the XSD type of this first parameter is `int`, hence the type of the function's parameter `itemID` is assumed to be `int`.
- To determine the return type of the operation, we observe that the variable `result` is returned, which has previously been assigned with the output of the

operation `submit`. In the XSD the according element (named `submitResponse`) is of type `boolean`. Hence, the return type of the function `purchase` is assumed to be `boolean`. Note that the example in hand contains only one `return` statement; in case a function contains several `return` statements, it becomes impracticable in general to determine the return type without advanced flow analysis.

For a more detailed discussion of the WSDL generation algorithm, specifically concerning the cases in which XSD types can or cannot be determined, see Subsection 5.3.3. The further improvement of this algorithm is part of our future work (see Section 7.1).

4.4.2 Dispatching Incoming Requests

Having generated and published the WSDL files of the protocol functions, clients invoke the PH to request protocol execution. Upon receiving a SOAP message the PH needs to determine the target SEPL document and the target function in this protocol. This is accomplished by an according WS-Addressing `Action` header to be sent in the SOAP message from the client: the format of the `Action` header is `<protocol name>:<function name>`.

```
1  <definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
2               xmlns:tns="http://infosys.tuwien.ac.at/dsg/daios/stateful"
3               xmlns:wsa="http://www.w3.org/2005/08/addressing" ...>
4    ...
5    <portType name="PortingProtocolPortType">
6      <operation name="port_numbers">
7        <input name="port_numbersRequest" message="tns:port_numbersRequest"
8            wsa:Action="PortingProtocol:port_numbers">
9        </input>
10       <output name="port_numbersResponse" message="tns:port_numbersResponse"
11           wsa:Action="PortingProtocol:port_numbers:response">
12       </output>
13     </operation>
14   </portType>
15   ...
16 </definitions>
```

Listing 9: WS-Addressing Action in Generated WSDL Document

Listing 9 contains an excerpt of the generated WSDL document for the porting service protocol introduced in Section 4. The according SEPL document contains one function `port_numbers` which is equal to the WSDL operation name. The input definition of this operation specifies the `Action PortingProtocol:port_numbers`. The operation's output specifies the same `Action` with a colon and the string `response` appended. WS-Addressing enabled SOAP clients will parse the WSDL and automatically append the according `Action` header (see Listing 10). `PortingProtocol` is the unique identifier under which the porting service protocol is registered in the PH. Neither this identifier nor the name of a function may contain a colon, hence the

`Action` string can easily be split at the colon by the PH to unambiguously determine which function it needs to execute.

```
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/"
          xmlns:tns="http://infosys.tuwien.ac.at/dsg/daios/stateful"
          xmlns:wsa="http://www.w3.org/2005/08/addressing" ...>
  <Header>
    <wsa:Action>PortingProtocol:port_numbers</wsa:Action>
    ...
  </Header>
  <Body>
    <tns:port_numbers>
      ...
    </tns:port_numbers>
  </Body>
</Envelope>
```

Listing 10: WS-Addressing Action in SOAP Invocation

4.4.3 Execution of the Target Protocol Function

After having determined the target protocol in the dispatching phase, the PH starts with its execution. Therefore it is necessary to convert the incoming SOAP message to objects which are interpretable by the SEPL script engine.

```
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/" ...>
  ...
  <Body>
    <tns:port_numbers>
      <username>telco0815</username>
      <password>secret</password>
      <request>
        ...
      </request>
      <request>
        ...
      </request>
    </tns:port_numbers>
  </Body>
</Envelope>
```

Listing 11: WS-Addressing Action in SOAP Invocation

A key aspect is to detect whether parameters have simple (`string`, `int`, ...), complex (XML markup) or array type. Consider the example SOAP invocation sketched in Listing 11. The Body element `port_numbers` contains four sub-elements: the strings `username` and `password` and two elements `request`. The PH detects that the two `request` elements are of the same type and make up an array. Eventually, the PH makes use of the SEPL client to execute the target SEPL function `port_numbers` with three parameters: the two strings and the array of requests.

During execution of the protocol the SOAP invocation to the PH blocks and returns the result when the SEPL client has finished. Further implementation details will be discussed in Section 5.

5 Implementation

In this section our prototype implementation of the SEPL framework is presented. Analogously to the Design section (Section 4), this section is divided into three subsections: one for each the SEPL client engine, the SEPL tools for model-driven development (SEPL code generator) and the SEPL protocol host server application. Firstly, we take a look at the "big picture", i.e., how the framework components are connected with each other. The details will be explained step by step later on. Figure 16 illustrates the implementation of the example scenario presented in Section 4. The scenario is based on the functionality of the `PortingService` 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 the *SOAP Client* (which represents the end-user) in the form of invokable WSDL operations. The PH is implemented as a Java Web Application [71] and is deployed in a Tomcat application server [2]. 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 (AD) models into SEPL code – a preprocessing step which is necessary in case the PortingService service is configured with an AD model. The code generation is optional but added here for the sake of completeness. If the end user wants to execute a protocol function, he uses a SOAP Client which

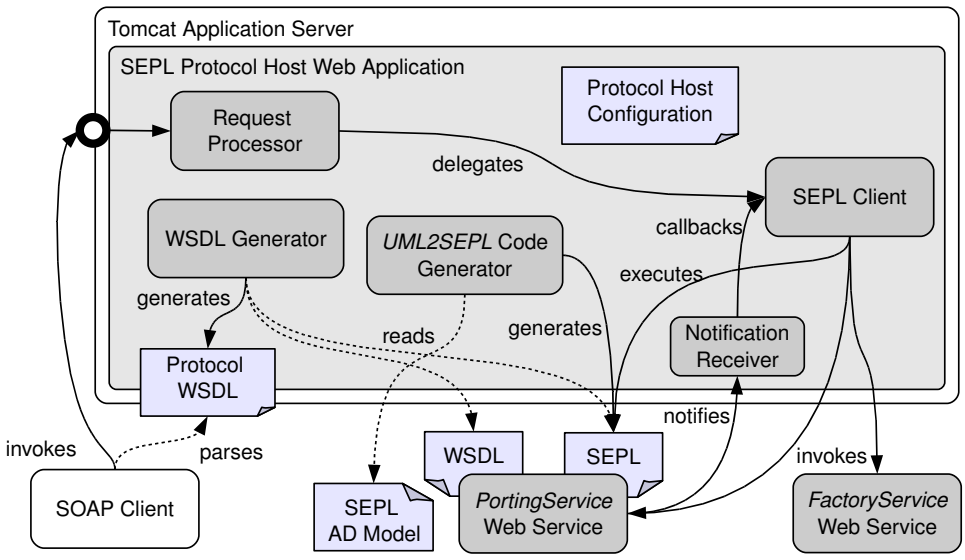


Figure 16: Connection Between the SEPL Framework Components

parses the *Protocol WSDL* and sends an according SOAP message to the network interface of the Tomcat server (illustrated as a thick circle). 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 it by invoking operations of the *FactoryService* and

the *PortingService* 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.

After this short example scenario discussion we turn to the details of the framework components in the following subsections.

5.1 SEPL Client Engine

The SEPL client (or SEPL *engine*) prototype implementation has been developed in the Java programming language (version 6). Figure 17 illustrates the structure of the engine and the responsibility of the separate parts, in the context of an example Web service and a service factory.

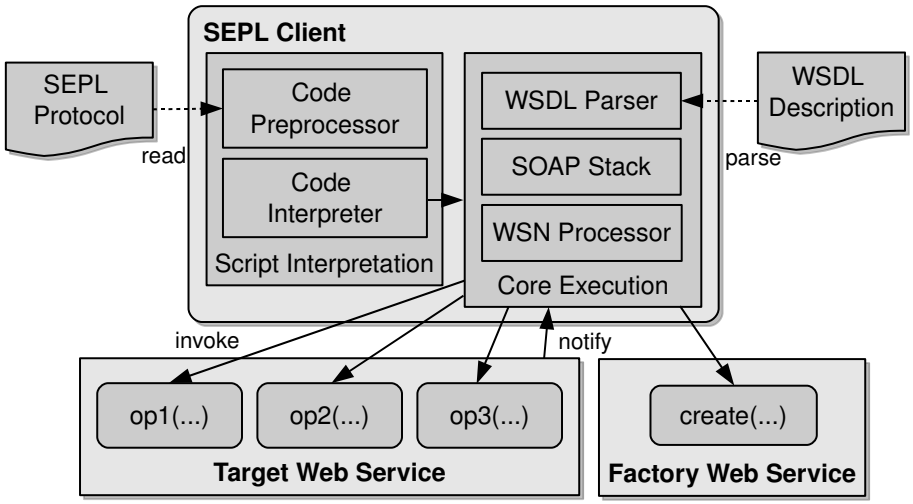


Figure 17: SEPL Engine Structure and Responsibilities

In preparation of SEPL documents for interpretation, the *Code Preprocessor* reads SEPL protocol files and applies certain modifications to convert SEPL code into the format of a concrete scripting language. 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. 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. 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*. The SOAP stack is based on the Daios framework. Listing 18 in the Appendix C contains Java source code which illustrates the usage of the SEPL client, and Listing 19 in the Appendix D contains an excerpt of the SEPL client implementation.

The **Code Interpreter** is based on the Java scripting engine *Pnuts* [9]. Each object occurring in SEPL code has its equivalent Java class. Pnuts is extensible in so far as it supports the definition of new object types which implement the interface `pnuts.lang.AbstractData`. The objects occurring in SEPL and their respective Java implementations for Pnuts are listed in Table 5.

SEPL object	Instance of class
`async`	`at.ac.tuwien.infosys.dsg.daios.stateful.pnuts.Async`
`factory`	`at.ac.tuwien.infosys.dsg.daios.stateful.pnuts.WSFactory`
`fault`	`at.ac.tuwien.infosys.dsg.daios.stateful.pnuts.SOAPFault`
`self`	`at.ac.tuwien.infosys.dsg.daios.stateful.pnuts.WSResource`
`properties`	`at.ac.tuwien.infosys.dsg.daios.stateful.pnuts.WSResourceProps`
Array object	`at.ac.tuwien.infosys.dsg.daios.stateful.pnuts.Array`
Callback object	`at.ac.tuwien.infosys.dsg.daios.stateful.pnuts.WSNCallback`
Daios message	`at.ac.tuwien.infosys.dsg.daios.stateful.pnuts.DaiosMessage`
XML structure	`at.ac.tuwien.infosys.dsg.daios.stateful.pnuts.XML`

Table 5: SEPL Objects and the Respective Java Classes for Pnuts

All classes listed in the table implement `pnuts.lang.AbstractData` and can be seen as the DSL-specific extension to the Pnuts core. Pnuts is responsible to parse and interpret SEPL code, the SEPL extension is responsible to perform domain-specific tasks such as Web service invocations, XML processing and so forth.

In order for the Pnuts core to interpret SEPL code, the source needs to be *preprocessed* by the **Code Preprocessor**. Pnuts cannot handle, for example, faults in the way they are syntactically defined in SEPL `catch`-blocks. The class `PnutsCodePreprocessor` makes all replacements and adjustments to the code source which are necessary in order for Pnuts to operate. Figure 18 depicts an example SEPL document before and after code preprocessing. The code preprocessor modifies the original document in various ways:

- At the beginning of the document, the necessary `import` statements are added. This makes the SEPL Pnuts classes available inside the code.
- The SOAP fault code `errorCode1` inside the `catch`-block needs to be transformed into a class definition, which is achieved as follows. A new Pnuts function named `Fault_errorCode1` gets included which defines the exception class `Fault_errorCode1` and returns a new instance of this exception. Due to the scoping of Pnuts, the defined exception class is also available in the function `xyz`, and hence the class can be used in the `catch` block. Whenever a SOAP

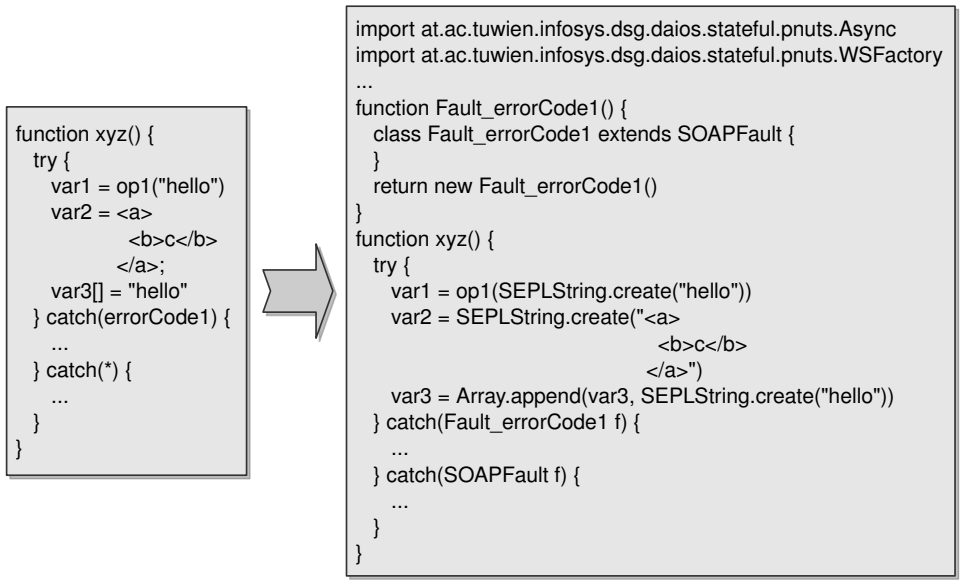


Figure 18: SEPL Code Preprocessing Example

fault with a `subCode` equal to `errorCode1` is received, the SEPL client will evaluate the expression `throw Fault_errorCode1()` (inside the function `xyz`). This throws a new fault which is caught by the embracing exception handler. Figure 18 illustrates how the code preprocessor alters the `catch`-block definitions with the according class types.

- The XML structure assigned to the variable `var2` is embraced with quotation marks.
- Every string `s` under quotation marks is replaced by `SEPLString.create(s)`. This static function determines the content of the string and returns 1) an object of type `XML` representing `s` if the string contains valid XML markup, or 2) the string `s` itself otherwise.
- The convenience SEPL expression `var3[] = "hello"`, which appends the string `"hello"` to the end of the array, is replaced by the expression `var3 = Array .append(var3, SEPLString.create("hello"))` which performs the according computation and returns the array.

5.2 SEPL Code Generator

In this section we briefly discuss the SEPL code generator prototype implementation [3]. The generator consists of two main parts: 1) a *UML factory* which reads encoded UML files and builds an in-memory object representation; 2) The actual *code writer*, which outputs the according SEPL code to an output stream. By making this distinction we strive for independence from the UML notation (the file format

[3] Classes are in package `at.ac.tuwien.infosys.dsg.daios.stateful.codegen`, though the package name of the classes mentioned in this section is not written out in full.

in which the UML AD is saved). New notations can be integrated by implementing an according factory class. Our implementation is based on the file format used by the *Eclipse modeling tools* [15] (MDT). With MDT, users can graphically design SEPL ADs in the Eclipse IDE and save the model to a file. The in-memory ob-

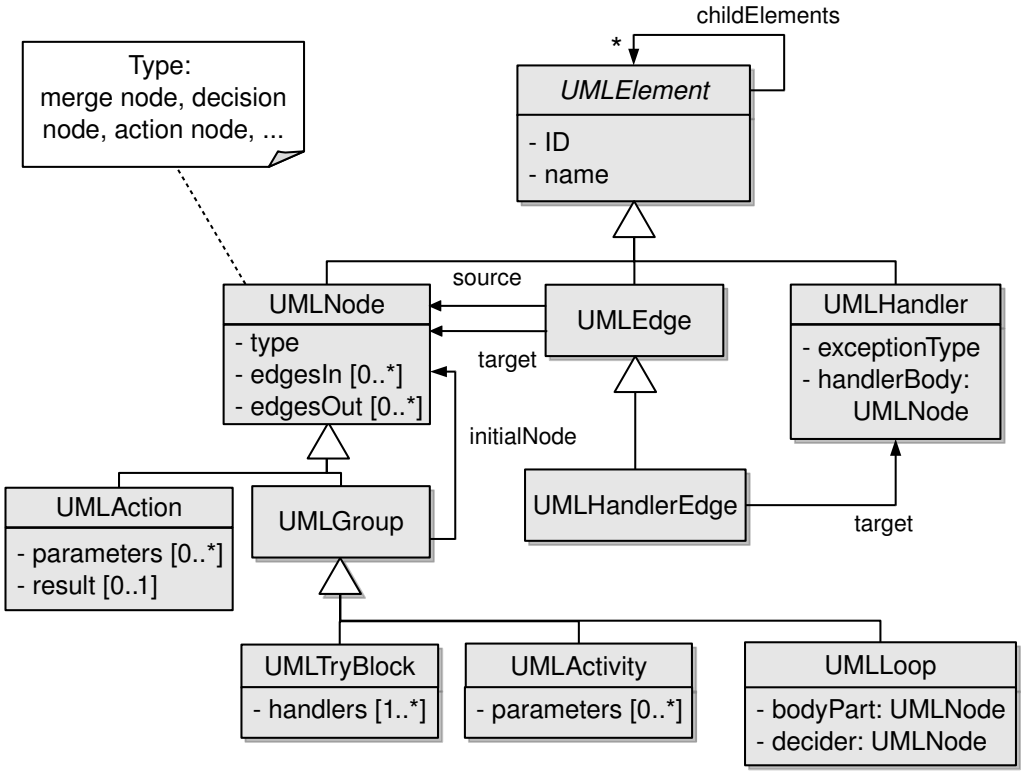


Figure 19: Class Diagram of the UML AD In-Memory Representation

ject representation of UML AD elements is presented in the class diagram in Figure 19. The abstract base class `UMLElement` holds an identifier (`ID`) and a `name` and, more importantly, contains a reflexive association `childElements` (*composite pattern*). `UMLNode` and `UMLEdge`, representing the basic elements of ADs, are direct subclasses of `UMLElement`. `UMLGroup`, a subclass of `UMLNode`, represents an aggregation of nodes and is subclassed by `UMLTryBlock`, `UMLActivity` and `UMLLoop`. `UMLAction` is the central element that represents all actions/operations in SEPL documents.

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. We will, however, take a short look at how the SEPL source code is actually produced from the in-memory model (Algorithm 5.2). The proceeding of the code writer is similar to the *visitor pattern* [21]: basically it iterates over all activities and calls the method `visit`, which recursively iterates over all nodes and outputs according SEPL code.

Algorithm 1 SEPL Code Writer Algorithm

```
1: lockedMergeNodes = ∅
2: for all activities a do
3:     VISIT(a)
4: end for
5: function VISIT(node)
6:     if node instance of UMLAction then
7:         output the action according to its type (e.g. invocation)
8:     else
9:         if node instance of UMLGroup then
10:            write "[function|while|for|try] <...> {" according to the type of node
11:            initialNode ← initial executable node of the group
12:            VISIT(initialNode)
13:            write "}"
14:        else if node is a DECISION node then
15:            m ← merge node connecting the paths of all outgoing edges of node
16:            lockedMergeNodes = lockedMergeNodes ∪ m
17:            for all outgoing edges e of node do
18:                write "[}else] [if(<condition>)]{" according to guard of e
19:                VISIT(e)
20:            end for
21:            lockedMergeNodes = lockedMergeNodes \ m
22:            write "}";
23:            VISIT(m) return
24:        else if node is a MERGE node and node ∈ lockedMergeNodes then
25:            return
26:        end if
27:    end if
28:    for all successor nodes s of node do
29:        VISIT(s)
30:    end for
31: end function
```

Inside of `visit`, the following is taken into consideration:

- Simple operations (`UMLActions`) are output according to their type, e.g. "<variable> = <operationName>(<parameters>)" for an invocation.
- For group nodes (`UMLGroups`), at first the "first line" of the group is written, e.g. "while(<condition>) {" for `while`-loops, or "try {" for `try`-blocks. Afterwards, a recursive call to `visit` is made with the `initial executable node` of this group. The initial node is either specified by a specific attribute such as `bodyPart` or it is the group's single node which has no incoming edges.
- A bit more complicated is the procedure to convert structures with decision and merge nodes into `if-else if-else` code structures. When the algorithm arrives at a decision node, it calls a sub-procedure which "travels" along all reachable paths and searches for the first common merge node (CMN) which can be reached on *all* paths. This merge node is considered the end of the `if-else if-else` structure. It is important to recapitulate that ADs must not contain cycles (see Subsection 4.3), because this simplifies the graph traversal as we do not need to keep track of nodes already visited. It is also possible that no

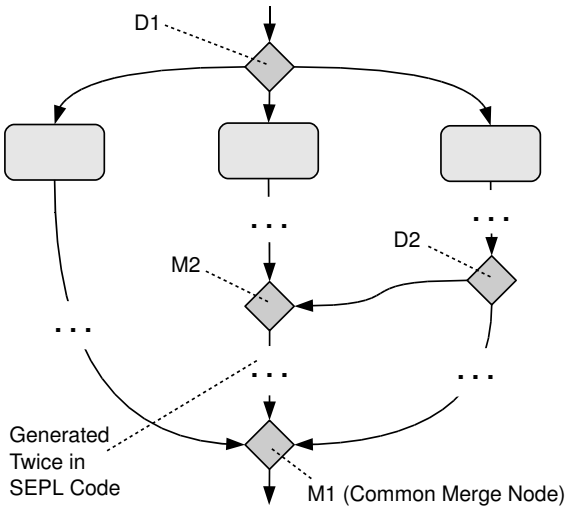


Figure 20: Common Merge Node and Intertwined Branches

such CMN exists (in those cases where the control flow does not continue after the `if-else if-else` structure). In this case the variable `m` in the algorithm is empty (`null`) and the call of `visit` with parameter `null` has no effects. Figure 20 illustrates the idea of the common merge node. The decision *D1* splits up the control flow into 3 branches. One branch will become the `if`-branch, the second will become the `else if`-branch and the third will become the `else`-branch in the generated SEPL code (note that edge guards are left out for simplicity). Whatever structures occur inside the three branches (indicated by three dots "..."), the branches are joined by the common merge node $M1$. Note that it is nevertheless possible to "intertwine" the branches in such manner that the control flow of one branch ramifies to the other branch (implemented using the

decision node $D2$ and the merge node $M2$). This is certainly not customary but can be very useful in some cases. The consequence of intertwined branches is that part of the code is generated twice in two different branches.

5.3 SEPL Protocol Host

In this subsection the crucial points of the SEPL Protocol Host (PH) implementation are explained.

5.3.1 Web Application Structure

Figure 21 contains a screenshot of the rough structure of the SEPL protocol host Web application. The SEPL protocol application is based on Apache AXIS2 [1]. But, unlike in AXIS2 where services are statically configured, SEPL configures all required Web services dynamically on deployment. The single most important file is `WEB-INF/classes/sepl.xml`, which contains the PH configuration. Every time a new PH is to be created, `sepl.xml` is the single file which needs to be adapted. The

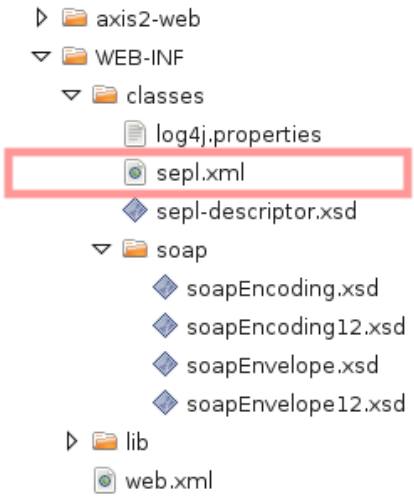


Figure 21: Web Application Structure

file `sepl-descriptor.xsd` is the XSD schema file for the configuration file `sepl.xml`. `WEB-INF/classes/soap/` contains XSD files required by the Daios framework. The directory `/WEB-INF/lib/` contains all required libraries, including those of Daios, SEPL and AXIS2. The Web application can be packed in a WAR [71, 72] file in exactly this structure and deployed to a Tomcat application server.

5.3.2 Configuration

The configuration file `sepl.xml` defines the parameters needed by the PH in order to work properly:

- `containerHost` and `containerPort`: hostname or IP address and port of the container in which the PH is deployed.
- `notificationServiceName`: name of the service used to receive notifications.
- `overwriteWSDLsOnStartup`: boolean value to indicate whether protocol WSDL files should be generated and overwritten on startup (deployment).
- `serviceProtocol`: element to define a protocol, its name and the location of the target WSDL and SEPL files. One or more protocols may be defined.

Listing 12 contains the configuration file used for our example scenario.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <sepl xmlns="http://infosys.tuwien.ac.at/dsg/sepl/descriptor"
3    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4    xsi:schemaLocation="http://infosys.tuwien.ac.at/dsg/sepl/descriptor
5    sepl-descriptor.xsd">
6
7    <containerHost>localhost</containerHost>
8    <containerPort>8080</containerPort>
9    <notificationServiceName>SeplNotificationService</notificationServiceName>
10   <overwriteWSDLsOnStartup>true</overwriteWSDLsOnStartup>
11
12   <serviceProtocol name="PortingProtocol">
13     <wsdlLocation>http://localhost:8080/Porting/Porting.wsdl</wsdlLocation>
14     <seplLocation>http://localhost:8080/Porting/Porting.sepl</seplLocation>
15   </serviceProtocol>
16
17 </sepl>
```

Listing 12: Example PH Configuration File sepl.xml

It contains the configuration of one protocol named `PortingProtocol`. The notification service is named `SeplNotificationService`. The Tomcat container is available under `localhost:8080`. Assuming the name of the WAR file is `sepl.war`, then the context path of the Web application (in Tomcat) is `/sepl` and the full URL of the notification service results in `http://localhost:8080/sepl/SeplNotificationService`.

5.3.3 Parameters Types and Return Types

Figure 22 illustrates - by means of a flowchart - 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 function's 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 (say, $var1$), we have to make another case distinction. If the

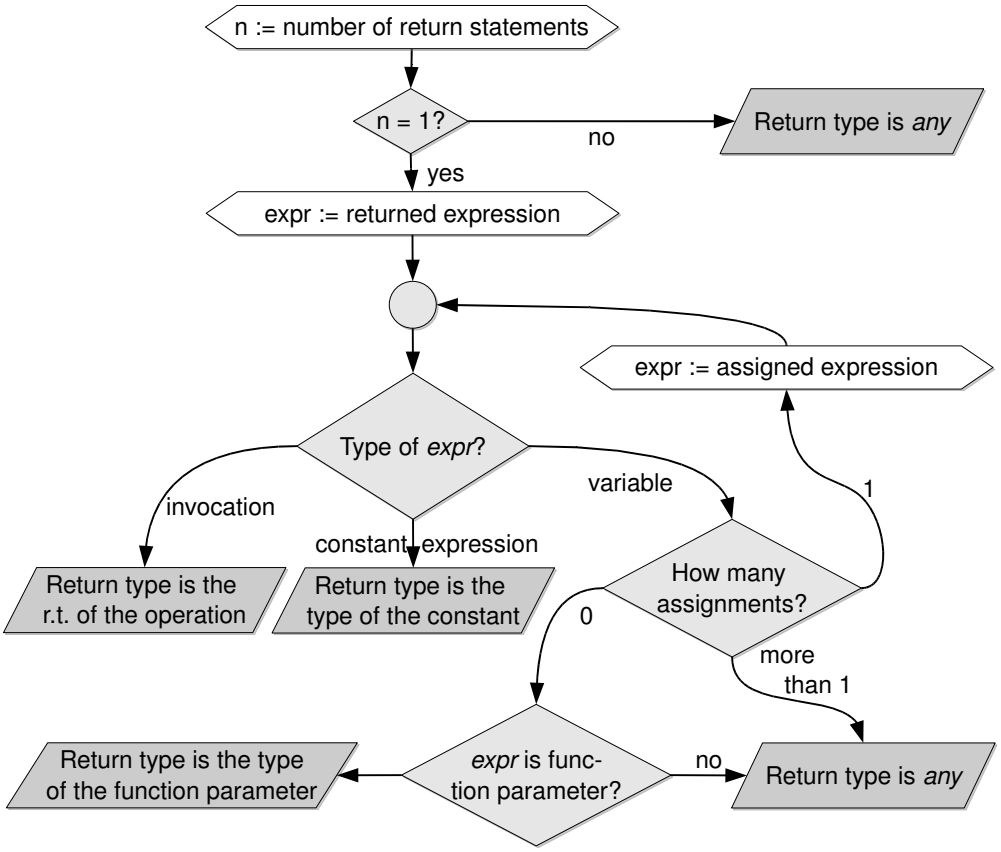


Figure 22: Determining the Return Type of a Function

variable gets assigned exactly once in the function (and assumingly *before* the return statement), a recursive call is 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. If this is the case, the return type of the function equals the type of the respective parameter.

Determining the **types of parameters** is less complicated. We mentioned in Subsection 4.2.2 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, it is rendered with XSD type `any` in the WSDL.

6 Evaluation

In this section we perform an evaluation of the work carried out in the thesis. The evaluation focuses on the design decisions discussed in Section 4 as well as the concrete prototype implementation presented in Section 5. The SEPL framework can hardly be compared to existing similar solutions on the whole. However, certain aspects of the framework can be compared to the related work: one aspect is how (time-)efficiently service protocols can be implemented using SEPL; a second aspect is the performance of executing a protocol functionality by means of the SEPL framework in comparison to other conceivable solutions. Framework-internally we evaluate the WSDL generation performance, which is a crucial factor for the startup speed of SEPL PH applications, for SEPL documents of various sizes.

6.1 Development Efficiency

In the following we consider the example number porting service protocol presented in Subsection 4.1 and compare the usage of the SEPL framework to other possible solutions for this problem with special regards to the development efficiency. We assume the existence of a service `PortingService` (`PS`) with operations `login`, `check_porting_status` and `schedule_porting_for`. The functionality of *MultiplePorting* is to perform number porting for a series of customers in one go – using the operations provided by the `PS` Web service. Principally, this scenario could be implemented in different ways:

- by creating a SEPL protocol file
- by creating a client which implements the business logic
- by creating a WS-BPEL process
- by extending the `PS` service itself (adding an additional operation).

Other solutions are conceivable, but we will focus on these 4 solutions. The SEPL implementation of the porting protocol has been printed in Listing 7 in Subsection 4.1. The code with its 23 lines seems slender and clear. To start the SEPL protocol execution, the five Java code lines in Figure 13 are sufficient.

```
1 Element[] requests = ...; // assume that the request array is given
2 SEPLClient client = new SEPLClient(
3     "http://localhost:8080/Porting/services/PortingService?wsdl",
4     new URL("http://localhost:8080/Porting/Porting.sepl").openStream() );
5 boolean ok = (Boolean)client.invoke("port_numbers", user, pass, requests);
```

Listing 13: Executing the Protocol With the SEPL Client

Implementing the protocol becomes more tedious when we use a pure Web service client library. Listing 14 illustrates how the *MultiplePorting* functionality can be implemented with the Daios `ServiceFrontend`. The source code gets blown up to more

than 60 lines. The code given in Listing 14 hides the asynchronous notification receiving, which takes place in a separate thread. We assume that received notifications are put to the thread-safe blocking queue `QUEUE`.

```
1  String WSDL_FACTORY = "...";
2  String WSDL_PORTING = "...";
3  String USERNAME = "...";
4  String PASSWORD = "...";
5  LinkedBlockingQueue QUEUE = ...; // reference to notification queue
6  String[][] requests = ...; // assume the requests are encoded as
7                             // a 2-dimensional String array
8
9  WSAEnabledInvokerFactory fac = new WSAEnabledInvokerFactory();
10 ServiceFrontend frontend = fac.createFrontend(
11   new URL(WSDL_PORTING), new URL(WSDL_FACTORY));
12
13 try {
14   DaiosMessage login = new DaiosMessage();
15   login.setString("username", USERNAME);
16   login.setString("password", PASSWORD);
17   frontend.setWSDLOperationName(new QName("login"));
18   DaiosMessage response = frontend.requestResponse(login);
19 } catch (InvocationException e) {
20   SOAPEncodedException e1 = (SOAPEncodedException)e.getOriginalException();
21   SOAPFault fault = new SOAPFault(e1.getElement());
22   return fault.getDetail();
23 }
24 List<String> requestCustomerList = new ArrayList<String>();
25 for(String[] req : requests) {
26   requestCustomerList.add(req[0]);
27   DaiosMessage check = new DaiosMessage();
28   check.set("customer", req[0]);
29   check.set("newProvider", req[1]);
30   frontend.setWSDLOperationName(new QName("check_porting_status"));
31   DaiosMessage status = frontend.requestResponse(check).getComplex("status");
32   if(!status.getBoolean("isPossible"))
33     throw new Fault(status.getString("problemDetail"));
34
35   DaiosMessage setCust = new DaiosMessage();
36   DaiosMessage update = new DaiosMessage();
37   update.setString("customer", req[0]);
38   setCust.set("Update", update);
39   frontend.setWSDLOperationName(new QName(NS_WSRP, "SetResourceProperties"));
40   frontend.requestResponse(setCust);
41   DaiosMessage setProv = new DaiosMessage();
42   update = new DaiosMessage();
43   update.setString("newProvider", req[1]);
44   setProv.set("Update", update);
45   frontend.requestResponse(setProv);
46
47   DaiosMessage schedule = new DaiosMessage();
48   schedule.set("time", req[2]);
49   frontend.setWSDLOperationName(new QName("schedule_porting_for"));
50   frontend.requestResponse(schedule);
51 }
52 List<DaiosMessage> results = new ArrayList<DaiosMessage>();
53 while(requestCustomerList.size() > 0) {
54   // notifications are received in a separate thread and put to QUEUE
55   DaiosMessage notif = QUEUE.take();
56   String customer = notif.getString("customer");
57   requestCustomerList.remove(customer);
58   results.add(notif);
59 }
```

```
60  frontend.setWSDLOperationName(new QName("Destroy"));
61  frontend.requestResponse(new DaiosMessage());
62  return results;
```

Listing 14: Number Porting Protocol Implemented Using Daios' `ServiceFrontend`

The source code grows even more if we implement the *MultiplePorting* functionality in WS-BPEL (see Listing 15). The design of BPEL requires developers to use many `assign` directives between operations. Even though some parts have been left out in the listing, we end up with more than 100 lines of BPEL code. Concerning the asynchronous invocation of `schedule_porting_for`, 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. In our case, we define the property `correlationID` to be the ID of the resource created by the factory when `createResource` is invoked. This ID is extracted 1) from the EPR returned by the factory service and 2) from the EPR contained in the notification message. Having these two values, the BPEL engine can assign notification messages to the correct process instances.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <process name="MultiplePorting" xmlns:tns="..." ...>
3   <!-- WSDL imports omitted -->
4   <partnerLinks>
5    <partnerLink name="processPL" myRole="MultiplePortingProtocol" .../>
6    <partnerLink name="targetService" partnerRole="MultiplePorting" .../>
7    <partnerLink name="factory" partnerRole="MultiplePortingFactory" .../>
8   </partnerLinks>
9   <variables>
10    <!-- variable declarations omitted -->
11  </variables>
12  <correlationSets>
13   <correlationSet name="CorrelationSet"
14                   properties="tns:correlationID">
15   </correlationSet>
16  </correlationSets>
17  <sequence>
18   <receive name="receive" createInstance="yes" partnerLink="processPL"
19    operation="port_numbers" portType="tns:Porting"
20    variable="Port_numbersIn"/>
21   <assign name="assignInitial">
22    <!-- omitted: initialization of input variables with empty literal -->
23   </assign>
24   <invoke name="create" partnerLink="factory" operation="createResource"
25     portType="tns:PortingFactoryPT" inputVariable="CreateResourceIn"
26     outputVariable="CreateResourceOut">
27    <correlations>
28     <correlation initiate="yes" set="CorrelationSet"/>
29    </correlations>
30   </invoke>
31   <assign name="assignEPR2">
32    <copy>
33     <from>$CreateResourceOut.parameters</from>
34      <to partnerLink="targetService"/>
35    </copy>
36    <!-- omitted: copy username and password to $LoginIn -->
37   </assign>
38   <scope name="LoginScope">
39    <faultHandlers>
```

```
40      <catch faultName="tns:InvalidCredentials">
41       <reply name="Reply2" partnerLink="processPL" operation="port_numbers"
42              portType="tns:MultiplePorting" variable="Port_numbersOut"/>
43      </catch>
44     </faultHandlers>
45     <invoke name="auth" portType="tns:PortingPT" partnerLink="targetService"
46       operation="login" inputVariable="LoginIn" outputVariable="LoginOut"/>
47    </scope>
48    <forEach name="ForEachRequest" parallel="no" counterName="i">
49     <startCounterValue>1</startCounterValue>
50     <finalCounterValue>count($Port_numbersIn.parameters/PortingRequest)
51     </finalCounterValue>
52     <scope name="RequestLoopScope">
53      <sequence name="Sequence1">
54       <!-- omitted: copy customer and newProvider from input (position
55            $i) to Check_statusIn and Perform_portingIn -->
56       <invoke name="chk" partnerLink="targetService"
57          portType="tns:PortingPT" operation="check_porting_status"
58          inputVariable="Check_statusIn" outputVariable="Check_statusOut"/>
59       <if name="check">
60        <condition>
61         not($Check_statusOut.parameters/status/tns:isPossible)
62        </condition>
63        <throw name="Throw" faultName="ns1:conflictingRequest"/>
64       </if>
65       <!-- omitted: copy process EPR address to $SubscribeIn -->
66       <invoke name="sub" partnerLink="targetService" operation="Subscribe"
67               portType="tns:PortingPT" inputVariable="SubscribeIn"
68               outputVariable="SubscribeOut"/>
69       <!-- omitted: copy customer to $SetResourcePropertiesIn -->
70       <invoke name="setCustomer" partnerLink="targetService"
71               operation="SetResourceProperties" portType="tns:PortingPT"
72               inputVariable="SetRPIn" outputVariable="SetRPOut"/>
73       <!-- omitted: copy newProvider to $SetRPIn -->
74       <invoke name="setNewProvider" partnerLink="targetService"
75               operation="SetResourceProperties" portType="tns:PortingPT"
76               inputVariable="SetRPIn" outputVariable="SetRPOut"/>
77       <!-- omitted: copy date to $ScheduleIn -->
78       <invoke name="schedulePorting" partnerLink="targetService"
79               operation="schedule_porting_for" portType="tns:PortingPT"
80               inputVariable="ScheduleIn" outputVariable="ScheduleOut"/>
81      </sequence>
82     </scope>
83    </forEach>
84    <forEach name="ForEachCallback" parallel="no" counterName="i">
85     <startCounterValue>1</startCounterValue>
86     <finalCounterValue>count($Port_numbersIn.parameters/PortingRequest)
87     </finalCounterValue>
88     <scope name="CallbackLoopScope">
89      <sequence name="Sequence2">
90       <receive name="notify" partnerLink="processPL" operation="Notify"
91                portType="tns:MultiplePorting" variable="NotifyIn">
92        <correlations>
93         <correlation set="CorrelationSet" initiate="no"/>
94        </correlations>
95       </receive>
96       <!-- omitted: append result to $Port_numbersOut array -->
97      </sequence>
98     </scope>
99    </forEach>
100   <invoke name="Destroy" partnerLink="targetService" operation="Destroy"
101           portType="tns:PortingPT" inputVariable="DestroyIn"
102           outputVariable="DestroyOut"/>
```

```
103     <reply name="return" partnerLink="processPL" operation="port_numbers"
104             portType="tns:MultiplePorting" variable="Port_numbersOut"/>
105   </sequence>
106 </process>
```

Listing 15: The *MultiplePorting* Protocol Implemented in WS-BPEL

The last of the mentioned possibilities to implement the *MultiplePorting* functionality is to extend the implementation of the `PS` service itself. Note that extending the service may even be impossible, e.g., because the service implementation is part of outdated legacy code and cannot be re-compiled. If we assume that the service is implemented in an (easily changeable) Java class, a new operation `port_numbers` can be added to the class which implements the protocol using direct (local) calls to the other service operations.

An overview of the comparison is printed in Table 6. Concerning the lines of code, SEPL clearly leads the list. In general, SEPL code is more light-weight and much less lines of code (LOC) are required compared to WS-BPEL documents or clients which implement the protocol functionality. The SEPL syntax is tailored to fast and simple programming, not necessarily requiring domain knowledge in Web service technology. The XML-based syntax of WS-BPEL is harder to read for humans, but XML is well suited to be processed by machines. Eventually, both BPEL and SEPL allow for graphical development using adequate modeling tools.

	LOC (ex.)	LOC (general)	Syntax	Graph. tools
SEPL	~ 20	small	scripting-like	✓
Client	~ 60	varies/medium	prog.lang.	×
WS-BPEL	> 100	huge	XML	✓
Service	-	varies/small	prog.lang.	×

Table 6: Comparison of Service Protocol Implementation Variants

6.2 Framework Performance

In the following we discuss the performance of the SEPL framework. Performance measurement tests have been carried out for the SEPL client engine and the WSDL generator. All tests have been run on a computer with AMD64 2.2GHz processor, 1GB RAM under the Linux operating system Ubuntu 8.04 [4] (Linux kernel 2.6.27-7).

6.2.1 SEPL Client Engine

The aim of the SEPL client engine tests is to determine how much overhead the (dynamic) interpretation of SEPL code causes, in comparison to (static) implementation

[4] http://www.ubuntu.com

of the protocol directly in the host programming language (Java) and in comparison to an implementation using WS-BPEL. The SEPL client engine utilizes the *Daios* framework, which has been measured against other popular Web service invocation frameworks with good results [35]. The aim of the tests presented here is *not* to repeat the comparison of Daios to other frameworks but to show the framework-internal overhead (mere Daios invocation time versus complete SEPL execution time) and the performance in comparison to WS-BPEL. We consider 2 example protocol functionalities in the tests (see Listing 16):

- `TestOperationChain` focuses on invoking many operations in a chain. The test invokes the test service's operation `testOp1` 100 times and uses the output of each invocation as the input to the next operation.
- `TestIOTransformation` focuses on the transformation of input and output data. The test performs 100 iterations where in each iteration the output of two operations (`testOp2_1` and `testOp2_2`) is merged to become the input of a third invocation (`testOp2_3`).

The SEPL implementation of the two example functions is printed in Listing 16. The protocol has been executed standalone using the SEPL client and hosted using the SEPL PH. Additionally to the SEPL implementation, we have developed a WS-BPEL process which executes the test functionalities. The process has been deployed in a Sun Glassfish [8] application server (version 2.1) with `sun-bpel-engine` [7] module. The fourth realization of the two example functionalities is a Java class implementation which relies solely on the Daios client and direct manipulation of `DaiosMessage` objects.

```
1  function TestOperationChain(input, iterations) {
2    result = testOp1(input)
3    i = 1
4    while(i < iterations) {
5      result = testOp1(result)
6      i = i + 1
7    }
8    return result
9  }
10
11 function TestIOTransformation(iterations) {
12   i = 0
13   while(i < iterations) {
14     result1 = testOp2_1()
15     result2 = testOp2_2()
16
17     both = "<bothResults><result1/><result2/></bothResults>"
18     both.result1 = result1.test1
19     both.result2 = result2.test2
20     testOp2_3(both)
21     i = i + 1
22   }
23 }
```

Listing 16: SEPL Performance Test Functions

For each of the four implementation versions (Daios, SEPL, SEPL PH and BPEL) [5], the tests have been performed in 13 iterations whereas the time was measured only in the last 10 iterations. The first 3 iterations were run without time measurement in order not to allow time-consuming initializations (initial parsing, object creation etc.) to distort the accuracy of the test results.

The results of the test runs are listed in Table 7. For both tests, the SEPL client and the execution using the SEPL PH were slightly slower than the Daios implementation and slightly faster than the WS-BPEL process. In the first test, the slowdown of *SEPL*

	Daios	SEPL	SEPL PH	BPEL
TestOperationChain				
Iterations	10	10	10	10
Total Duration (ms)	3573	3772	4229	4753
Avg. Duration (ms)	357.3	377.2	422.9	475.3
Std. Deviation	46.90	51.75	81.79	44.36
Increase compared to *Daios*	-	5.57%	18.36%	33.03%
Increase compared to *SEPL*	-	-	12.12%	26.01%
Increase compared to *SEPL PH*	-	-	-	12.39%
Value of `t` against *Daios*	-	0.901 (`I`)	2.2 (`S`)	5.78 (`S`)
Value of `t` against *SEPL*		-	1.493 (`I`)	4.551 (`S`)
Value of `t` against *SEPL PH*			-	1.781 (`S`)
Comparison value `t(0.95,18)`			1.734	
TestIOTransformation				
Iterations	10	10	10	10
Total Duration (ms)	9529	11941	12292	12413
Avg. Duration (ms)	952.9	1174.1	1229.2	1241.3
Std. Deviation	68.46	115.08	176.51	31.93
Increase compared to *Daios*	-	25.31%	28.99%	30.27%
Increase compared to *SEPL*	-	-	2.94%	3.95%
Increase compared to *SEPL PH*	-	-	-	0.98%
Value of `t` against *Daios*	-	5.224 (`S`)	4.615 (`S`)	12.073 (`S`)
Value of `t` against *SEPL*		-	0.827 (`I`)	1.779 (`S`)
Value of `t` against *SEPL PH*			-	0.213 (`I`)
Comparison value `t(0.95,18)`			1.734	

Table 7: Performance Test Results

amounts to only `5.57%` compared to *Daios*. In the second test, where more input-output transformations are performed, the slowdown amounts to roughly a quarter `25.31%`. The *SEPL PH* execution durations are slightly higher than the durations for *SEPL*, due to the time consumed by messages exchanged with the PH. *SEPL* and *SEPL PH* are faster than *BPEL* on average, although the difference in the second

[5]The source code of the implementation with Daios and the WS-BPEL process source code are not included in this document for the sake of brevity.

test is very small (`3.95%` and `0.98%`). The durations measured for *SEPL* and *SEPL PH* have a higher standard deviation than both *Daios* and *BPEL*.

To check the significance of the results, we performed a *t-test* [36] calculation. For each combination (A, B) of implementation versions we formulate the null hypothesis $H_0 = time(A) < time(B)$ (A executes the protocol faster than B) and 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 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* table [37]. 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`. In all cases where the t is greater than or equal to `1.734`, the hypothesis H_0 cannot be rejected and we conclude that A runs *significantly* faster than B. In Table 7 this is indicated with the capital letter `S` in brackets. If the value of `t` is smaller than `1.734`, we reject the null hypothesis H_0 and assume the opposite, i.e., that the run time difference between A and B is *insignificant* (indicated with the capital letter `I`). In the `TestOperationChain` test, *BPEL* is significantly slower than all other variants. In `TestIOTransformation`, *Daios* is significantly faster than all other variants. In both cases, *SEPL PH* is significantly slower than *Daios*, but insignificantly slower than *SEPL*. For both tests, *SEPL* is significantly faster than *BPEL*.

6.2.2 WSDL Generator

In this section we discuss the runtime performance of the SEPL WSDL generator. Upon startup of the managing application container, the SEPL PH starts initialization of all protocols that it has been configured with. This involves generating the WSDL documents which describe the interface of the protocol functions hosted by the PH. Time is a crucial factor when it comes to this initialization and we want to check how well the WSDL generator scales for large service protocols and large numbers of SEPL functions.

First of all, we analyze which factors influence the duration of the WSDL generation process. The total time T_W it takes to generate one protocol WSDL document from the SEPL file and the WSDL file of the target service depends mainly 1) on the time needed to preprocess the SEPL code in order to make it interpretable by the Pnuts parser (T_C), 2) on the time consumption of the Pnuts parser which creates an in-memory representation of the SEPL source code (T_P), 3) on the duration between requesting and having fully parsed the target WSDL file (T_T) and 4) on the "net" WSDL generation time, i.e., the time needed to combine all the collected information and to write out the WSDL as a Java String (T_N). This relation is summarized in Table 8.

$T_W = T_C + T_P + T_T + T_N$	
T_W	WSDL generation time
T_C	Code preprocessing time
T_P	Pnuts parsing time
T_T	Target service WSDL parsing time
T_N	Net WSDL generation time

Table 8: WSDL Generation Time Formula

In the class `WSDLGenPerformanceTest` we have set up an appropriate test environment: the SEPL function `port_numbers` (see Subsection 4.2.1) is duplicated N times with names `port_numbers1, port_numbers2, ..., port_numbersN` and N steadily increasing ($N \in \{1, 2, 3, 5, 10, 15, 25, 35, 50, 65, 80, 100\}$). The resulting SEPL documents, together with the WSDL file of the number porting service, are used as input to the WSDL generator. In the source code, we use an *interceptor* approach to measure the required durations. At each the start and the end of an activity, the interceptor is called to save the current timestamp (e.g. `Interceptor.event(EventType.START_PNUTS_PARSE)`, `Interceptor.event(EventType.FINISH_PNUTS_PARSE)`). At the end of the computation, the difference between the timestamps is calculated. The tests have been run 10 times and the arithmetic mean has been calculated. Figure 23 depicts the results of this benchmark. We can see from the figure that T_P and

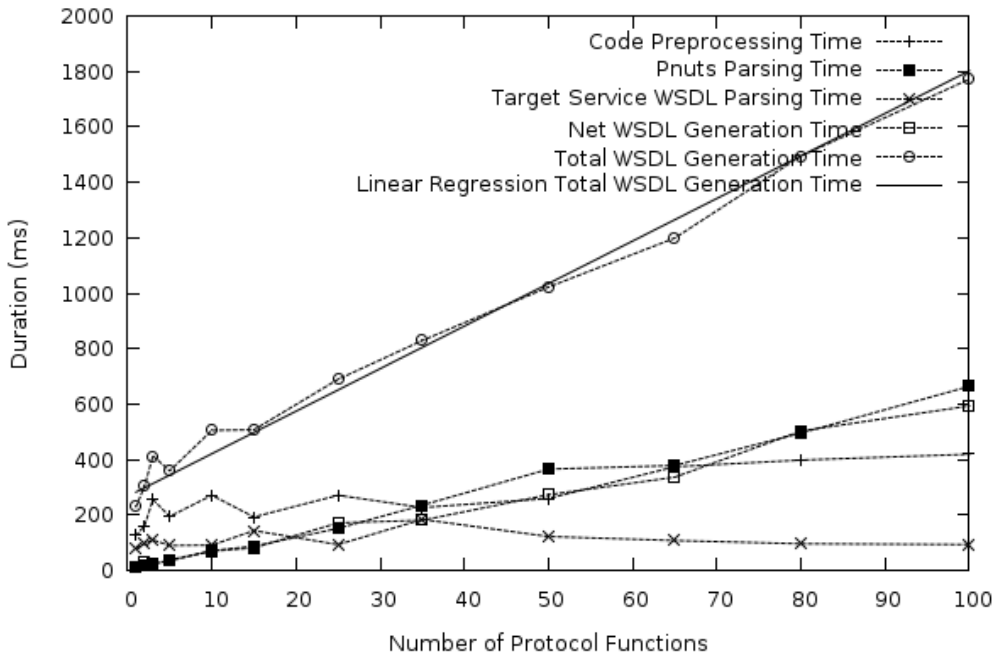


Figure 23: WSDL Generator Benchmark Results

T_N mount near-linearly at almost the same rate. The duration T_T is near-constant with increasing number of SEPL functions, whereas the code preprocessing time T_C slightly increases. The solid line in the figure depicts the linear regression curve of the total WSDL generation time: `duration = 267.9 + 15.3 * #(functions)`,

which fits with a standard deviation of (only) `39.296`. Hence we can estimate that the WSDL generation takes a base time of less than `300ms` and an additional time of roughly `15ms` per protocol function. However, this is only a rough estimation and the generation time, in general, depends on the size and structure of the protocol functions as well as the number of parameters of the single functions.

7 Conclusion and Future Work

Even though part of the SOA community have argued that Web services are inherently stateless, stateful services have become widespread and popular. The Web Services Resource Framework is a set of specifications that has emerged from the need for a unified way to describe, address, create, invoke and destroy stateful service instances. The statefulness of a service imposes constraints on the way and the order in which messages are to be exchanged. We speak of the *intra-service protocol* to express the dynamic interface of a service, knowledge of which – additionally to the static interface in the form of the WSDL definitions – is a prerequisite in order for clients to successfully interact with the service. The intra-service protocol specifies exactly the anticipated behavior (the input data that is required, the order in which the operations have to be performed and the way in which the output of an operation has to be transformed to serve as the input to another operation in order to achieve a certain functionality) as well as the unexpected behavior of the service (handling and raising of SOAP Faults). Service protocol specification is similar to service composition in so far as new functionalities are composed, but with the major difference that only one service is considered. Currently there is still an evident lack for an effective way to express and execute such protocols.

In this thesis we have addressed the problem of intra-service protocol specification and execution and have come up with an appropriate solution, the SEPL framework. SEPL is a DSL with a scripting-language like syntax whose features to specify service protocols range from basic control flow directives and synchronous/asynchronous invocations to fault handling and easy access to WS-Resource properties and elements in XML markup. Based on the definition of the DSL, we have presented the design and implementation of the three main components of the SEPL framework: the SEPL execution engine (SEPL client), the UML-based SEPL development environment and the SEPL protocol host (SEPL server). The SEPL client has a clear interface and is ready to be used in third party Java applications. The performance evaluation has shown that the protocol execution is performed with a minor overhead compared to static implementation of the protocol's business logic. The SEPL code generator facilitates model-driven development of service protocols on the basis of UML activity diagrams. The presented UML activity diagram models have the same expressive power as SEPL documents and the advantage that existing graphical tools (Eclipse Modeling Framework) provide a clear presentation to both technical specialists and non-experts. The SEPL protocol host offers a convenient way to expose SEPL protocol functions as Web service operations, to execute SEPL protocols centrally on a server machine and thereby to take away the protocol execution responsibility from clients. The proposed WAR file structure and the configuration mechanism allows for the efficient deployment of protocol hosts in standard J2EE application servers such as Apache Tomcat. In the evaluation it became apparent that SEPL fosters efficient development of service protocols to create new service functionalities based on existing Web service implementations.

7.1 Future Work

Even though the presented SEPL framework implementation is fully functioning in its current state, the development is not considered finished. We plan to extend the SEPL architecture by new features and to introduce further improvements to the current implementation:

- The WSDL generation algorithm will be enhanced to reach a maximum level of information concerning the XSD types of messages in the generated WSDL documents. In Section 5 we mentioned that the algorithms to determine return types and parameter types do not handle special cases such as two `return` statements occurring in a SEPL function. Using an advanced data-flow analysis it will become possible to handle even complex cases and to provide a near-complete picture of the interface of SEPL functions in generated WSDL documents without use of the XSD type `any`.
- We plan to extend the SEPL syntax by capabilities for parallel execution of activities. Parallelism can drastically reduce the net time consumed by a protocol execution and is also an integral part of service composition languages such as WS-BPEL (`flow` instruction). Parallelism is well supported in Java in terms of synchronized multi-threading and therefore straight-forward to implement in the execution engine of the SEPL client.
- WSDL documents have emerged as the general purpose way of describing Web services. Due to its popularity and extensibility, WSDL is often used to carry additional information such as semantic annotations [91] besides the mere syntactical rules for the message exchange with a service. We plan to develop a suitable way to include SEPL protocols in the WSDL definition of services, either by a link to a SEPL file or by embedding the SEPL code directly. Another thinkable method to communicate SEPL protocols to service consumers is the usage of *WS-MetadataExchange* [17].
- To identify single conversations with target services, the SEPL framework currently relies on WS-Addressing and the factory-instance pattern: For each protocol execution, a new service instance is created and its EPR is used in subsequent invocations to the service. We plan to provide more flexibility in this matter and support the use of conversation IDs in the SOAP header of messages exchanged between the SEPL client and the target service.
- Besides the use of WS-BaseNotification messages, we will analyze and implement different methods of asynchronous service invocations. From the current point of view, we mainly consider the *Callback Pattern* of *WS-MessageDelivery* [85].

Appendix

A List of Abbreviations

AD	Activity Diagram
B2B	Business-to-Business
BPEL	(Web Services) Business Process Execution Language
CMN	Common Merge Node
CPO	Cell Phone Operator
Daios	Dynamic and asynchronous invocation of services
DO	Data Object
DSL	Domain Specific Language
EPR	Endpoint Reference
HTTP	Hypertext Transfer Protocol
ID	Identifier
IDE	Integrated Development Environment
IT	Information Technology
JMS	Java Message Service
LOC	Lines Of Code
MEP	Message Exchange Pattern
MDA	Model Driven Architecture
MDD	Model Driven Development
MDT	Eclipse Model Development Tools
OASIS	Organization for the Advancement of Structured Information Standards
OGSA	Open Grid Services Architecture
OGSI	Open Grid Services Infrastructure
OMG	Object Management Group
OOP	Object-Oriented Programming
OWL-S	Web Ontology Language for Web Services
PH	Protocol Host
PN	Petri Net
RPC	Remote Procedure Call
SDE	Service Data Element
SEPL	Service Protocol Language
SMTP	Simple Mail Transfer Protocol
SOA	Service-Oriented Architecture
SOAP	Simple Object Access Protocol
SoC	Service-oriented Computing
UDDI	Universal Description, Discovery and Integration
UML	Unified Modeling Language
URI	Universal Resource Identifier
URL	Universal Resource Locator
WS-BPEL	Web Service Business Process Execution Language

Continued on Next Page...

WS-CDL	Web Service Choreography Description Language
WS-Resource	Web Service Resource
WSA	Web Services Addressing
WSCI	Web Service Choreography Interface
WSCL	Web Services Conversation Language
WSDL	Web Service Definition Language
WSDL-S	Web Service Semantics
WSMO	Web Service Modeling Ontology
WSRF	Web Services Resource Framework
WSRP	Web Services Resource Properties
WAR	Web Application Archive
WWW	World Wide Web
XMI	XML Metadata Interchange
XML	eXtensible Markup Language
XPath	XML Path Language
XSD	XML Schema Definition

Table 9: List of Abbreviations

B SEPL Syntax Rules

```
1  PROTOCOL = {ASSIGNMENT} FUNCTION {FUNCTION} ;
2  FUNCTION = "function" IDENTIFIER FUNC_PARAMETERS Eol BLOCK ;
3  EOL = "\r" | "\n" | "\r\n" ;
4  Eol = {EOL} ;
5  FUNC_PARAMETERS = "(" [ FUNC_PARAM { "," FUNC_PARAM } ] ")" ;
6  FUNC_PARAM = Eol IDENTIFIER Eol ;
7  BLOCK = BLOCK2 | EXPRESSION {";" [ EXPRESSION ] } ;
8  BLOCK2 = "{" Eol [ EXPRESSION_LIST ] "}" ;
9  EXPRESSION_LIST = EXPRESSION { (";" | EOL) [ EXPRESSION ] } ;
10 EXPRESSION = ASSIGNMENT | STATEMENT_EXPRESSION ;
11 ASSIGNMENT = IDENTIFIER "=" ASSIGNABLE ;
12 ASSIGNABLE = INVOCATION | CONSTRUCTOR | (XML ";") | PRIMARY_EXPR;
13 STATEMENT_EXPRESSION = IF_STATEMENT | WHILE_STATEMENT | DO_STATEMENT |
14                        FOR_STATEMENT | "break" | "continue" | RETURN |
15                        INVOCATION | TRY_STATEMENT | "throw" EXPRESSION ;
16 IF_STATEMENT = "if" Eol "(" Eol EXPRESSION Eol ")" Eol
17                BLOCK {ELSEIF_NODE} [ELSE_NODE] ;
18 ELSEIF_NODE = Eol "else" "if" "(" Eol EXPRESSION Eol ")" Eol BLOCK ;
19 ELSE_NODE = Eol "else" Eol BLOCK ;
20 WHILE_STATEMENT = "while" Eol "(" Eol EXPRESSION Eol ")" Eol BLOCK ;
21 TRY_STATEMENT = "try" Eol BLOCK2 { Eol CATCH_BLOCK } [ Eol FINALLY_BLOCK ] ;
22 CATCH_BLOCK =  "catch" "(" IDENTIFIER ")" Eol BLOCK2 ;
23 FINALLY_BLOCK = "finally" Eol BLOCK2 ;
24 DO_STATEMENT = "do" Eol BLOCK2 Eol "while" Eol "(" Eol EXPRESSION Eol ")" ;
25 FOR_STATEMENT = "for" Eol "(" Eol (IDENTIFIER Eol ":" Eol EXPRESSION Eol |
26                                    {ASSIGNMENT} ";" Eol [CONDITION Eol] ";"
27                                       Eol {ASSIGNMENT} ) ")" Eol BLOCK ;
28 RETURN = "return" [ASSIGNABLE] ;
29 NUMBER = INTEGER | FLOATING_POINT ;
30 DIGIT = "0".."9" ;
31 INTEGER =  DIGIT { DIGIT };
32 FLOATING_POINT = INTEGER "." INTEGER [EXPONENT] | INTEGER EXPONENT ;
33 EXPONENT = ("e"|"E") ["+"|"-"] INTEGER ;
34 LETTER = "a".."z" | "A".."Z" | "_" ;
35 IDENTIFIER = LETTER { LETTER | DIGIT } ;
36 NOT_QUOTATION_MARK = "\x0020".."\x0021" | "\x0023".."\xffff" ;
37 STRING = "\"" {NOT_QUOTATION_MARK | "\\\""} "\"" ;
38 STRING_ANY = {NOT_QUOTATION_MARK | "\""} ;
39 NCNAME = IDENTIFIER [ {IDENTIFIER | "-"} IDENTIFIER ] ;
40 XML = "<" [ NCNAME ":" ] NCNAME
41          { [ NCNAME ":" ] NCNAME "=" STRING } ">"
42            Eol (XML | STRING_ANY) Eol "</" [ NCNAME ":" ] NCNAME ">" ;
43 PRIMARY_EXPR = STRING | NUMBER | IDENTIFIER | MATH_EXPRESSION;
44 INVOCATION = IDENTIFIER "(" PARAMETERS ")" ;
45 PARAMETERS = "(" [ PARAM { "," PARAM } ] ")" ;
46 PARAM = IDENTIFIER | STRING | MATH_EXPRESSION ;
47 MATH_EXPRESSION = STRING | NUMBER | IDENTIFIER |
48                    UNARY_EXPRESSION | BINARY_EXPRESSION |
49                    "(" UNARY_EXPRESSION ")" | "(" BINARY_EXPRESSION ")" ;
50 UNARY_EXPRESSION = ("!" | "-" | "~") MATH_EXPRESSION ;
51 BINARY_OPERATOR = ("+"|"-"|"*"|"/"|"%"|"<"|"<="|">"|">="|"||"|"&&") ;
52 BINARY_EXPRESSION = MATH_EXPRESSION BINARY_OPERATOR MATH_EXPRESSION;
53 CONSTRUCTOR = IDENTIFIER "(" PARAMETERS ")" ;
54 CONDITION = MATH_EXPRESSION ;
```

Listing 17: SEPL Syntax Rules in EBNF

C Usage of the SEPL Client

```
1  import at.ac.tuwien.infosys.dsg.daios.stateful.Util;
2  ...
3  SEPLClient c = new SEPLClient(
4      ClientTest.class.getResourceAsStream("Porting.sepl"),
5      "127.0.0.1",
6      8090);
7
8  long time1 = new Date().getTime() + 10*1000;
9  long time2 = new Date().getTime() + 13*1000;
10 String requests =
11 "<requests>" +
12  "<request>" +
13   "<customer>customer</customer>" +
14   "<newProvider>orangeTel</newProvider>" +
15   "<time>" + time1 + "</time>" +
16  "</request>" +
17  "<request>" +
18   "<customer>customer2</customer>" +
19   "<newProvider>blueTel</newProvider>" +
20   "<time>" + time2 + "</time>" +
21  "</request>" +
22 "</requests>";
23
24 Element reqs = Util.toElement(requests);
25 Object result = c.invoke("port_numbers",
26      "whummer",
27      "secretPassword",
28      Util.getChildElements(reqs).toArray() );
29
30 System.out.println("Protocol execution result: " + result);
```

Listing 18: Using the SEPL Client to Execute the Number Porting Protocol

D SEPL Client Implementation

```
1  public class SEPLClient {
2
3    private Pnuts pnuts;
4    private String source;
5    private Context context;
6    private WSNotificationReceiver wsnReceiver;
7    private WSResource service;
8    private HttpServer server;
9    private boolean clientIsServerCreator;
10   private boolean hasBeenShutDown;
11   private final String protocolExecutionID = UUID.randomUUID().toString();
12   private CodePreprocessor preprocessor = new PnutsCodePreprocessor();
13   ...
14
15   private void initialize(String wsdlURL, InputStream is, HttpServer serv,
16                           WSNotificationReceiver recv) throws Exception {
17     ...
18   }
19
20   public Object execute(String protocolFunction, Object ... args)
21                 throws ProtocolExecutionFault {
22     try {
23       hasBeenShutDown = false;
24       if(server != null) {
25         server.start();
26       }
27       PnutsFunction func = null;
28       try {
29         func = (PnutsFunction)context.resolveSymbol(protocolFunction);
30         if(func == null)
31           func = (PnutsFunction)context.getId(protocolFunction);
32       } catch (Exception e) {
33         logger.error(e);
34       }
35       if(func == null)
36         throw new SEPLException("Unknown function: " + protocolFunction);
37       for(int i = 0; i < args.length; i++)
38         args[i] = Util.toSEPLObject(args[i]);
39       Object result = func.call(args, context);
40       if(result instanceof XML) {
41         List<Element> elements = ((XML)result).getElementsList();
42         result = elements.toArray(new Element[]{});
43         if(elements.size() == 1)
44           result = elements.get(0);
45       }
46       return result;
47     } catch(PnutsException e) {
48       throw new ProtocolExecutionFault(e.getThrowable());
49     } finally {
50       shutDown();
51     }
52   }
53   ...
54 }
```

Listing 19: Operation `execute` in the Class `SEPLClient`

References

[1] Apache Software Foundation. Apache Axis 2. http://ws.apache.org/axis2/. Visited: 2008-09-27.

[2] Apache Software Foundation. Apache Tomcat. http://tomcat.apache.org. Visited: 2008-11-01.

[3] Colin Atkinson and Thomas Kuhne. Model-driven development: a metamodeling foundation. *Software, IEEE*, 20(5):36–41, Sept.-Oct. 2003.

[4] Daniela Berardi, Fabio De Rosa, Luca De Santis, and Massimo Mecella. Finite State Automata As Conceptual Model For E-Services. *Journal of Integrated Design and Process Science*, 8(2):105–121, 2004.

[5] Tim Berners-Lee, Roy Fielding, and Larry Masinter. RFC 3986, Uniform Resource Identifier (URI): Generic Syntax. http://rfc.net/rfc3986.html, 2005. Visited: 2008-12-13.

[6] Russel Butek. Which style of WSDL should I use? http://www.ibm.com/developerworks/webservices/library/ws-whichwsdl/. Visited: 2008-10-02.

[7] CollabNet Corporation. BPEL Service Engine User's Guide. https://open-esb.dev.java.net/kb/60/ep-bpel-se.html. Visited: 2009-02-25.

[8] CollabNet Corporation. Glassfish - Open Source Application Server. https://glassfish.dev.java.net. Visited: 2008-11-01.

[9] CollabNet Corporation. The Pnuts language. https://pnuts.dev.java.net/. Visited: 2008-11-02.

[10] Microsoft Corporation. XLANG/s Language. http://msdn.microsoft.com/en-us/library/aa577463.aspx. Visited: 2009-01-21.

[11] Francisco Curbera, Matthew Duftler, Rania Khalaf, William Nagy, Nirmal Mukhi, and Sanjiva Weerawarana. Unraveling the Web Services Web: An Introduction to SOAP, WSDL, and UDDI. *IEEE Internet Computing*, 6(2), 2002.

[12] John Davies, Rudi Studer, and Paul Warren. *Semantic Web Technologies: Trends and Research in Ontology-based Systems*. John Wiley & Sons, 2006.

[13] Schahram Dustdar and Wolfgang Schreiner. A survey on web services composition. *International Journal of Web and Grid Services*, 1(1):1–30, 2005.

[14] Eclipse Foundation. Eclipse BPEL Project. http://www.eclipse.org/bpel/. Visited: 2009-02-15.

[15] Eclipse Foundation. Model Development Tools (MDT). http://www.eclipse.org/uml2. Visited: 2008-11-01.

[16] Thomas Erl. *Service-Oriented Architecture. Concepts, Technology, and Design*. Prentice Hall, 2005.

[17] Keith Ballinger et al. Web Services Metadata Exchange (WS-MetadataExchange). `http://specs.xmlsoap.org/ws/2004/09/mex/WS-MetadataExchange.pdf`, August 2006.

[18] Roy Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter, Peter Leach, and Tim Berners-Lee. RFC 2616, Hypertext Transfer Protocol – HTTP/1.1. `http://www.rfc.net/rfc2616.html`, 1999. Visited: 2009-02-03.

[19] Daniela Florescu, Andreas Grünhagen, and Donald Kossmann. XL: an XML programming language for Web service specification and composition. *Computer Networks*, 42(5):641–660, 2003.

[20] Ian Foster and Carl Kesselman. *The Grid 2: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003.

[21] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[22] Globus Alliance. The WS-Resource Framework. `www.globus.org/wsrf/specs/ws-wsrf.pdf`, 2004. Visited: 2008-07-27.

[23] Karl Gottschalk, Stephen Graham, Heather Kreger, and James Snell. Introduction to Web services Architecture. *IBM Systems Journal*, 41(2), 2002.

[24] Roy Grønmo and Ida Solheim. Towards Modeling Web Service Composition in UML. In *Proceedings of the 2nd International Workshop on Web Services: Modeling, Architecture and Infrastructure*, pages 72–86, 2004.

[25] Rachid Hamadi and Boualem Benatallah. A Petri net-based model for web service composition. In *ADC '03: Proceedings of the 14th Australasian database conference*, pages 191–200, Darlinghurst, Australia, 2003. Australian Computer Society, Inc.

[26] Carsten Hentrich and Uwe Zdun. Patterns for process-oriented integration in service-oriented architectures. In *Proceedings of 11th European Conference on Pattern Languages of Programs (EuroPlop 06)*, 2006.

[27] Yousra BenDaly Hlaoui and Leila Jemni BenAyed. Toward an UML-based composition of grid services workflows. In *AUPC '08: Proceedings of the 2nd international workshop on Agent-oriented software engineering challenges for ubiquitous and pervasive computing*, pages 21–28, New York, NY, USA, 2008. ACM.

[28] Michael N. Huhns and Munindar P. Singh. Service-Oriented Computing: Key Concepts and Principles. *IEEE Internet Computing*, 9(1), 2005.

[29] International Organization for Standardization. ISO/IEC 14977:1996 - Extended BNF. `http://standards.iso.org/ittf/PubliclyAvailableStandards/s026153_ISO_IEC_14977_1996(E).zip`, 1996.

[30] Hu Jingjing, Zhao Xing, Cao Yuanda, and Zhou Ruitao. A Service Composition Model with Characteristic of Transaction Based on Finite State Machine. *Computer and Electrical Engineering, 2008. ICCEE 2008. International Conference on*, pages 450–454, Dec. 2008.

[31] Json.org. Introducing JSON. `http://www.json.org/`. Visited: 2009-02-14.

[32] Gerhard Kramler, Elisabeth Kapsammer, Werner Retschitzegger, and Gerti Kappel. Towards Using UML 2 for Modelling Web Service Collaboration Protocols. In *Interoperability of Enterprise Software and Applications*, pages 227–238. Springer London, 2006.

[33] Vinay Kulkarni and Sreedhar Reddy. Separation of concerns in model-driven development. *Software, IEEE*, 20(5):64–69, Sept.-Oct. 2003.

[34] Philipp Leitner. The Daios Framework - Dynamic, Asynchronous and Message-oriented Invocation of Web Services. Master's thesis, Vienna University of Technology, http://www.infosys.tuwien.ac.at/Staff/leitner/dowloads/masterthesis.pdf, 2007.

[35] Philipp Leitner, Florian Rosenberg, and Schahram Dustdar. DAIOS - Efficient Dynamic Web Service Invocation. *To appear in IEEE Internet Computing (regular paper)*, 2009.

[36] Richard Lowry. Critical Values of t. `http://faculty.vassar.edu/lowry/ch11pt1.html`. Visited: 2009-03-05.

[37] Richard Lowry. t-Test for the Significance of the Difference between the Means of Two Independent Samples. `http://faculty.vassar.edu/lowry/apx_c.html`. Visited: 2009-03-05.

[38] Anton Michlmayr, Florian Rosenberg, Christian Platzer, Martin Treiber, and Schahram Dustdar. Towards recovering the broken SOA triangle: a software engineering perspective. In *IW-SOSWE '07: 2nd international workshop on Service oriented software engineering*, pages 22–28, New York, NY, USA, 2007. ACM.

[39] Robin Milner. *Communicating and Mobile Systems: The Pi Calculus*. Cambridge University Press, 1999.

[40] NetBeans Community. Developer Guide to the BPEL Designer. `http://www.netbeans.org/kb/60/soa/bpel-guide.html`. Visited: 2009-02-15.

[41] Eric Newcomer and Greg Lomow. *Understanding SOA with Web Services*. Addison Wesley Professional, 2004.

[42] Jörg Nitzsche, Tammo van Lessen, and Frank Leymann. WSDL 2.0 Message Exchange Patterns: Limitations and Opportunities. In *ICIW '08: Proceedings of the 2008 Third International Conference on Internet and Web Applications and Services*, pages 168–173, Washington, DC, USA, 2008. IEEE Computer Society.

[43] Object Management Group. OMG Model Driven Architecture. http://www.omg.org/mda/. Visited: 2009-01-21.

[44] Object Management Group. OMG Unified Modeling Language (OMG UML), Superstructure, V2.1.2. http://www.omg.org/spec/UML/2.1.2/Superstructure/PDF/. Visited: 2008-11-01.

[45] Object Management Group. OMG Unified Modeling Language Specification. http://www.omg.org/spec/UML/1.5/PDF/. Visited: 2009-02-12.

[46] Object Management Group. Unified Modeling Language (UML). http://www.uml.org/.

[47] Object Management Group. Object Constraint Language. http://www.omg.org/docs/formal/06-05-01.pdf, 2006.

[48] 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. Visited: 2009-02-14.

[49] Open Grid Forum. Open Grid Services Infrastructure (OGSI), Version 1.0. http://www.ggf.org/documents/GFD.15.pdf. Visited: 2008-11-02.

[50] Open Grid Forum. The Open Grid Services Architecture, Version 1.5. http://www.ogf.org/documents/GFD.80.pdf. Visited: 2008-11-02.

[51] Oracle Corporation. Oracle BPEL Process Manager. http://www.oracle.com/technology/products/ias/bpel/index.html. Visited: 2009-02-15.

[52] Organization for the Advancement of Structured Information Standards (OASIS). OASIS Web Services Security (WSS) TC. http://www.oasis-open.org/committees/wss/. Visited: 2009-02-17.

[53] Organization for the Advancement of Structured Information Standards (OASIS). Web Services Context Specification. http://docs.oasis-open.org/ws-caf/ws-context/v1.0/wsctx.html.

[54] Organization for the Advancement of Structured Information Standards (OASIS). Web Services Resource Lifetime 1.2 (WS-ResourceLifetime). http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-ResourceLifetime-1.2-draft-03.pdf, 2004.

[55] 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. Visited: 2009-02-14.

[56] 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.

[57] Organization for the Advancement of Structured Information Standards (OASIS). Web Services Resource 1.2 (WS-Resource). `http://docs.oasis-open.org/wsrf/wsrf-ws_resource-1.2-spec-os.pdf`, 2006. Visited: 2008-07-27.

[58] Organization for the Advancement of Structured Information Standards (OASIS). Web Services Resource Framework. `http://www.oasis-open.org/committees/wsrf`, 2006. Visited: 2008-09-27.

[59] 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. Visited: 2008-07-27.

[60] Bart Orriëns, Jian Yang, and Mike P. Papazoglou. *Service-Oriented Computing - ICSOC 2003*, volume 2910/2003, chapter Model Driven Service Composition, pages 75–90. Springer Berlin / Heidelberg, 2003.

[61] Mike P. Papazoglou. Service-Oriented Computing: Concepts, Characteristics and Directions. In *WISE '03: Proceedings of the Fourth International Conference on Web Information Systems Engineering*, page 3, Washington, DC, USA, 2003. IEEE Computer Society.

[62] Savas Parastatidis and Jim Webber. CSP SSDL Protocol Framework. `http://www.ssdl.org/docs/v1.3/html/CSP%20SSDL%20Protocol%20Framework%20v1.3.html`, 2005.

[63] Savas Parastatidis and Jim Webber. MEP SSDL Protocol Framework. `http://www.ssdl.org/docs/v1.3/html/MEP%20SSDL%20Protocol%20Framework%20v1.3.html`, 2005.

[64] Savas Parastatidis and Jim Webber. Rules-based SSDL Protocol Framework. `http://www.ssdl.org/docs/v1.3/html/Rules%20SSDL%20Protocol%20Framework%20v1.3.html`, 2005.

[65] Savas Parastatidis and Jim Webber. The SOAP Service Description Language. `http://www.ssdl.org/docs/v1.3/html/SSDL%20v1.3.html`, 2005.

[66] Savas Parastatidis, Simon Woodman, Jim Webber, Dean Kuo, and Paul Greenfield. Asynchronous Messaging between Web Services Using SSDL. *IEEE Internet Computing*, 10(1):26–39, 2006.

[67] Carl Adam Petri. *Kommunikation mit Automaten.* PhD thesis, University of Bonn, Germany, 1962. (In German).

[68] Dumitru Roman, Uwe Keller, Holger Lausen, Jos de Bruijn, Rubén Lara, Michael Stollberg, Axel Polleres, Cristina Feier, Cristoph Bussler, and Dieter Fensel. Web service modeling ontology. *Appl. Ontol.*, 1(1):77–106, 2005.

[69] Tony Spiteri Staines. Intuitive Mapping of UML 2 Activity Diagrams into Fundamental Modeling Concept Petri Net Diagrams and Colored Petri Nets. In *ECBS '08: Proceedings of the 15th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems*, pages 191–200, Washington, DC, USA, 2008. IEEE Computer Society.

[70] Sun Microsystems Inc. Java SE Technologies - Database. `http://java.sun.com/javase/technologies/database/`. Visited: 2009-02-15.

[71] Sun Microsystems Inc. JSR-000154 JavaTM Servlet 2.4 Specification. `http://jcp.org/aboutJava/communityprocess/final/jsr154/`. Visited: 2009-02-14.

[72] Sun Microsystems Inc. What's New in Java Servlet API 2.2? `http://java.sun.com/developer/technicalArticles/Servlets/servletapi/`. Visited: 2009-02-25.

[73] UDDI.org. UDDI Technical White Paper. `http://www.uddi.org/pubs/Iru_UDDI_Technical_White_Paper.pdf`, 2000. Visited: 2008-07-31.

[74] World Wide Web Consortium (W3C). Extensible Markup Language (XML). `http://www.w3.org/XML/`. Visited: 2009-01-21.

[75] World Wide Web Consortium (W3C). Web Service Choreography Interface (WSCI) 1.0. `http://www.w3.org/TR/wsci/`. Visited: 2009-01-21.

[76] World Wide Web Consortium (W3C). Web Services Activity. `http://www.w3.org/2002/ws/`. Visited: 2009-02-14.

[77] World Wide Web Consortium (W3C). XML Path Language (XPath). `http://www.w3.org/TR/xpath/`, 1999.

[78] World Wide Web Consortium (W3C). Simple Object Access Protocol (SOAP) 1.1. `http://www.w3.org/TR/soap11/`, 2000. Visited: 2009-02-14.

[79] World Wide Web Consortium (W3C). Web Services Architecture. `http://www.w3.org/TR/2002/WD-ws-arch-20021114/`, 2002. Visited: 2009-02-17.

[80] World Wide Web Consortium (W3C). Web Services Conversation Language (WSCL) 1.0. `http://www.w3.org/TR/wscl10/`, 2002. Visited: 2009-01-21.

[81] World Wide Web Consortium (W3C). WSDL, Web Service Description Language. `http://www.w3.org/TR/wsdl`, 2002. Visited: 2008-10-21.

[82] World Wide Web Consortium (W3C). SOAP Version 1.2 Part0: Primer. `http://www.w3.org/TR/soap12-part0/`, 2003. Visited: 2008-09-24.

[83] World Wide Web Consortium (W3C). OWL Web Ontology Language. `http://www.w3.org/TR/owl-features/`, 2004. Visited: 2009-01-21.

[84] World Wide Web Consortium (W3C). Web Services Addressing (WS-Addressing). http://www.w3.org/Submission/ws-addressing/, 2004. Visited: 2009-01-21.

[85] World Wide Web Consortium (W3C). WS-MessageDelivery Version 1.0. http://www.w3.org/Submission/ws-messagedelivery/, 2004. Visited: 2009-02-17.

[86] World Wide Web Consortium (W3C). XML Schema Part 1: Structures Second Edition. http://www.w3.org/TR/xmlschema-1/, 2004. Visited: 2009-02-17.

[87] World Wide Web Consortium (W3C). XML Schema Part 2: Datatypes Second Edition. http://www.w3.org/TR/xmlschema-2/, 2004. Visited: 2009-02-17.

[88] World Wide Web Consortium (W3C). Web Service Semantics - WSDL-S. http://www.w3.org/Submission/WSDL-S/, 2005. Visited: 2009-01-21.

[89] World Wide Web Consortium (W3C). Web Services Description Language (WSDL) Version 2.0 Part 0: Primer - W3C Candidate Recommendation 27 March 2006. http://www.w3.org/TR/2006/CR-wsdl20-primer-20060327/, 2006.

[90] World Wide Web Consortium (W3C). Web Services Policy 1.2 - Framework (WS-Policy). http://www.w3.org/Submission/WS-Policy/, 2006. Visited: 2009-02-03.

[91] World Wide Web Consortium (W3C). Semantic Annotations for WSDL and XML Schema. http://specs.xmlsoap.org/ws/2004/09/mex/WS-MetadataExchange.pdf, August 2007.

[92] World Wide Web Consortium (W3C). SOAP Version 1.2 Part 1: Messaging Framework (Second Edition). http://www.w3.org/TR/soap12/, 2007. Visited: 2009-02-14.

[93] World Wide Web Consortium (W3C). XQuery 1.0: An XML Query Language. http://www.w3.org/TR/xquery/, 2007.

[94] Web service interoperability organization (WS-I). Basic Profile Version 1.2. http://www.ws-i.org/Profiles/BasicProfile-1.2.html. Visited: 2007-08-02.

[95] Simon Woodman, Savas Parastatidis, and Jim Webber. Sequencing Constraints SSDL Protocol Framework. http://www.ssdl.org/docs/v1.3/html/SC%20SSDL%20Protocol%20Framework%20v1.3.html, 2005.

[96] World Wide Web Consortium (W3C). Web Services Choreography Description Language Version 1.0, W3C Working Draft. http://www.w3.org/TR/ws-cdl-10, 2004. Visited: 2007-07-31.

[97] Uwe Zdun, Markus Voelter, and Michael Kircher. Pattern-Based Design of an Asynchronous Invocation Framework for Web Services. *International Journal of Web Services Research*, 1(3):42–62, 2004.

[98] Jia Zhang, Carl K. Chang, Jen-Yao Chung, and Seong W. Kim. WS-Net: a Petri-net based specification model for Web services. In *Proceedings of the IEEE International Conference on Web Services (ICWS)*, 2004.