Context Constraint Integration and Validation in Dynamic Web Service Compositions

Claus Pahl, MingXue Wang, and Kosala Yapa Bandara

School of Computing, Dublin City University
Dublin, Ireland
[cpahl|mwang|kyapa]@computing.dcu.ie

Abstract. System architectures that cross organisational boundaries are usually implemented based on Web service technologies due to their inherent interoperability benefits. With increasing flexibility requirements, such as on-demand service provision, a dynamic approach to service architecture focussing on composition at runtime is needed. The possibility of technical faults, but also violations of functional and semantic constraints require a comprehensive notion of context that captures composition-relevant aspects. Context-aware techniques are consequently required to support constraint validation for dynamic service composition. We present techniques to respond to problems occurring during the execution of dynamically composed Web services implemented in WS-BPEL. A notion of context – covering physical and contractual faults and violations – is used to safeguard composed service executions dynamically. Our aim is to present an architectural framework from an application-oriented perspective, addressing practical considerations of a technical framework.

1 Introduction

System architectures that cross organisational boundaries are usually implemented based on Web service technologies due to their inherent interoperability benefits. With increasing flexibility requirements, such as on-demand service provision, a dynamic approach to service architecture focussing on composition at runtime is needed. The possibility of technical faults, but also violations of functional and semantic constraints require a comprehensive notion of context that captures composition-relevant aspects. Contractual definitions reflect the needs of partners – flexibility to deal with these constraints at the level of contracts and service-level agreements is essential in dynamic, on-demand applications. The physical environment needs to be monitored as faults can be caused by device, platform and other technical factors. Context-aware techniques are required to support constraint validation and fault management for dynamic service composition. A flexible solution is sought that guarantees that these constraints are satisfied.

We present techniques to respond proactively to problems occurring during the execution of dynamically composed Web services implemented in WS-BPEL (or BPEL for short) [7]. A notion of context – covering physical and contractual faults and violations – is used to safeguard composed service executions dynamically. In particular, we introduce the following aspects:

– context ontology: an ontology-based context model that integrates business and technology context aspects,

– context constraint integration: we add monitoring instrumentation to BPEL using a context model-driven constraint validation,

– constraint monitoring and fault handling: use of BPEL fault handling to capture context constraint violations and faults at runtime,

– constraint violation analysis: implementation of recovery and remedial strategies using intelligent mapping of faults to context aspects.

The techniques can realise central functions of a middleware platform for dynamic and context-adaptive Web services. Our aim is to present an architectural framework from an application-oriented perspective, addressing practical considerations of a technical framework. We illustrate the benefits of the proposed technologies through an electronic payments application.

We discuss principles of service composition and specifically dynamic composition and process execution in Section 2. Then, the core techniques are introduced: a context ontology in Section 3, the constraint integration technique in Section 4, an introduction of fault tolerance in Section 5, a monitoring and fault handling technique in Section 6, and a fault analysis approach in Section 7. An evaluation of the technical framework in terms of reliability and performance is provided in Section 8. Finally, we discuss trends and current issues before ending with some conclusions.

2 Dynamic Service Composition

We discuss different aspects and stages in the autonomic composition of Web services. These aspects and stages are supported by the individual components of the overall architecture, see Fig. 1.

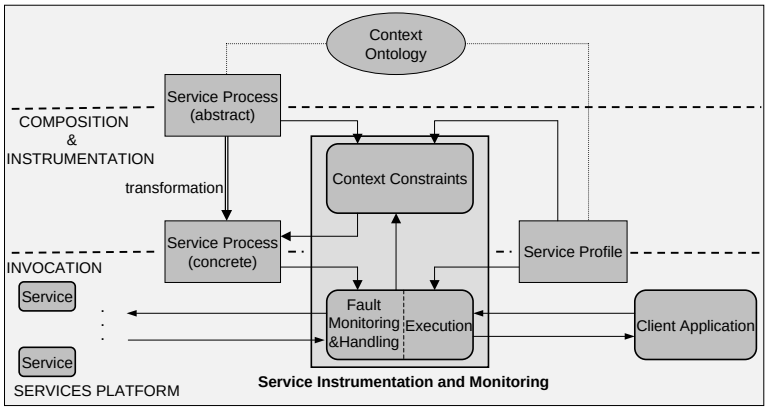

Fig. 1. System architecture

2.1 Context and Constraints

The autonomic composition of Web services usually starts with a planning process based on an abstract goal [13]. This approach allows a planner to consider a set of loosely coupled goals as a planning problem. An abstract composition plan is produced from the goals, from which an executable service process can be derived. Plan generation forms the starting point for the integration of context constraints. Context is the sum of all factors that can influence dynamic composition. While functional aspects are often considered during planning and composition, some aspects such as quality can only be determined and validated during execution. We call these dynamically validated aspects context constraints, or constraints for short.

We present a context ontology capturing a wide range of aspects of a service in relation to its environment – which we define as its context. Context constraints to be validated are generated (possibly based on a contract or service-level agreement between user and provider) and linked to constraint checkers, which validate the constraint during service process execution. Data collectors are used to determine actual context attributes dynamically. Constraint validation is woven into the BPEL application process.

2.2 Fault Monitoring and Fault Handling

Faults can occur during the execution of service processes – as a consequence of technical runtime problems such as the unavailability of external services or the violation of contractual constraints captured in the context model. Our context model captures and integrates functional, quality, domain and technical runtime environment aspects. Fault monitoring is responsible for fault (and constraint violation) detection and data collection. A fault is an abnormal condition or defect that may lead to failure.

BPEL allows to catch and manage faults using fault handlers. Fault handlers can be attached to an entire process or smaller execution scopes. If the process or scope terminates normally, the attached fault handlers get ignored, but if a fault occurs, it is propagated to the fault handler. Using BPEL's fault handlers for monitoring can avoid overheads on additional supervision monitoring process and the BPEL engine-dependent monitoring component.

2.3 Case Study

Our case study focuses on a broker architecture where a client can request utility bills from a range of devices. The service broker (e.g. a bank) is responsible for providing the requested utility bill in the requested currency to the requested device. We assume that both user and service provider have registered with the service broker. This example illustrates the effect of context on local and external services in a composed service process.

An initial user request is a goal that results in a dynamic generation of a service process, composing the application Web services and weaving in context-dependent constraint validation services. Constraints and their validation support are generated based on context information of the service involved. The user calls the `UserBillRequest` at the service broker. This process is composed of `ProviderBillRequest`, `ProviderBillResponse` and `UserBillResponse` application services. Initially, the request is internally analysed, then each provider of services is contacted (invocation of

external services to proved bill responses), and finally, the response is adapted to the needs of the end user.

All related context constraints are integrated into the Web service process as pre-conditions or post-conditions, i.e. all context constraints are grouped under these two categories by a context constraint generator. The bill format may vary depending on the destination context. The user might expect a bill on her/his mobile device (e.g. user-friendly format in appropriate resolution) whereas the service broker expects it in machine-processable format (e.g. XML). Fault monitoring – based on BPEL's fault handlers that capture constraint violations – is the start of an analysis process that determines a remedial strategy.

3 Context Ontology for Service Composition

3.1 Context Model

Context has recently been explored in many projects to facilitate the development and deployment of context-aware and adaptable Web services [19]. Wang et al. propose CONON for modelling context in pervasive computing environments, identifying location, user, activity and computational entities as fundamental context categories [30]. Doulkeridis et al. define two types of context called service context and user context for mobile services [10]. The service context implies the location of the service, its version, the providers identity, the type of the returned results and its cost of use. The user context characterizes the users current situation including location, time, temporal constraints, device capabilities and user preferences. Hong et al. propose a context-aware learning architecture ontology for ubiquitous learning environments defining context in four top-level classes as Person, Place, Activity and Computational Entities [11]. Often, location is the central context concern.

In order to determine a context model for autonomic services composition, we followed an empirical approach by looking at a number of case studies. Three case study scenarios have been defined in three domains illustrating the needs of a complete and flexible context model and applicable context-determined services. The scenarios – a traditional financial service (billing and payment), an e-learning application and convenience services – were analysed. In these case studies, context stems from different domains, ranging from classical business scenarios to modern services and convenience infrastructures. The important context aspects often vary significantly from application to application and some context aspects emerge more frequently such as device type or security. We organize these context concerns in a comprehensive and extensible model to capture context for Web services composition.

We derived a context model ontology. Four major context categories are identified in the proposed context ontology as Functional Context, which is useful in autonomic services composition in general; Quality of Service Context, which is useful in achieving dynamic composition; Domain Context, which is useful in achieving autonomic composition in different organizations; and Platform Context, which captures the technical environment. Together, the four categories capture all aspects (knowledge) of potential relevance for dynamic composition. The syntactical aspects of the service interface are part of this knowledge, as is device or domain-specific knowledge.

Brahim and Yacine have proposed a context categorization and context matching approach for Web services [14]. However, a lack of integrative context models that can be used in autonomic service composition led us to develop our context model for changing service environments.

3.2 The Service Composition Context Ontology

Based on our empirical observations, we define context as any static or dynamic client, provider or service related information, which enables or enhances efficient integration of clients, providers and services. Services need to be aware of their context if they were to automatically adapt to changing circumstances. A number of individual context categories are defined in the context ontology. These are grouped into four top-level context areas: functional, quality, domain, and platform. Our aim is to be comprehensive, i.e. to embrace the functional focus of planning and composition, but also the device and location focus of many current context notions.

Functional Context: This describes the operational features of services. The notion of functional context in Web services is sub-grouped:

- Syntax: includes input/output parameters that define operations, messages, data types of the parameters for invoking a service.
- Effect: includes the pre-conditions and post-conditions, i.e. the operational effect of an operation execution.
- Protocol: refers to a consistent exchange of messages among services involved in services composition to achieve their goals. It includes context on conversation rules and data flow.

Quality of Service Context (QoS): Qualitative properties can be organized into four groups [15] of quantifiable attributes based on the type of measurement performed by each attribute [21].

- Runtime Attributes: relates to the execution of a service. Performance; the measurement of the time behaviour of services in terms of response time, throughput etc. Reliability; the ability of a service to be executed within the maximum expected time frame. Availability; the probability that the service is accessible.
- Business Attributes: assess a service from a business perspective. Cost; the price for execution. Reputation; measures the services trustworthiness. Regulatory; a measure of how well a service is aligned with government or organizational regulations.
- Security Attributes: describe whether the service is compliant with security requirements. Integrity; protecting information from being deleted or altered. Authentication; ensure that both consumers and providers are identified and verified. Non-repudiation; ability of the receiver to prove to a third party that the sender did send a message. Confidentiality; protecting information from being read by anyone not authorized.
- Trust Attributes: refer to establishment of trust relationships between client and providers – a combination of technical assertions (measurable and verifiable quality) and relationship-based factors (reputation, history of cooperation).

Other quality of service attributes or groups can be added to these four fundamental groups.

Domain Context: Each application domain may need its own context (locale) for interacting with services:

– Semantic: refers to semantic framework (i.e. concepts and their properties) in terms of vocabularies, taxonomies or ontologies.

– Linguistic: language used to express queries, functionality and responses.

– Measures and Standards: refers to locally used standards for measurements, currencies, etc.

Platform Context: The technical environment a service is executed in.

– Device: refers to the computer/hardware platform on which the service is provided.

– Connectivity: refers to the network infrastructure used by the service to communicate.

3.3 Context Ontology Construction

The starting point for the ontology construction are existing service specifications. WSDL provides the syntactic input; semantic service ontologies like OWL-S or WSMO provide further functional and non-functional aspects. Often, formally or informally described service-level agreements provide other details such as domain and platform-specific aspects.

Our context ontology is a knowledge framework to capture composition-relevant information. It is not meant to replace existing descriptions. Ontology mappings can be defined between service and context ontologies. Context model information comes from very different sources. The functional and quality contexts are derived from service descriptions; platform contexts are captured based on system and platform data. Domain context is based on external information sources such as domain models or external settings (like languages or units). This diversity requires an integrating framework, which we provide in the form of a context model ontology. These context instances are modelled into a single context ontology based on OWL [8]. The context model ontology is an OWL-DL ontology that, at its core, captures the context model categories in the format of a taxonomy (concept level of the ontology) [27]. We only illustrate a few excerpts of the context ontology with Manchester OWL syntax. For instance, we define:

```
Class:        FunctionalContext            Class:        Syntax
SubClassOf:   Context                      SubClassOf:   FunctionalContext
DisjointWith: QoS or Domain or Platform    DisjointWith: Effect or Protocol
```

Specific links – e.g. between Trust and Security – can be formalised by using `SubClassOf` instead of `DisjointWith` to define trust as a specific computer security aspect. Specific properties can be formulated, for instance `Syntax hasInterface MIN 1 and hasInterface SOME string`, which requires a syntax element to have at least one interface of type string associated to it.

3.4 Context Constraints

Context model instances express concrete requirements, for instance, concrete values for expected response times. Context reasoning for generation is used in two forms: checking the consistency of context and deducing context constraints based on defined context properties. For instance, when

a client makes a request about her gas bill from her mobile device, an abstract process (plan) is generated to fulfil her request. A context constraints list is generated based on the agreements made between the parties participating in fulfilling the client's request. This list contains context instances of context types used in agreements and also instances of deduced contexts types – e.g. Mobile Phone is a context instance of device type and Bill Format is a deduced context type for the device type.

We capture concrete constraints as context model instances. We illustrate this using service `UserBillRequest` with parameters `UserID, UserName, UserAddress, UtilityType, BillRequestDevice, BillRequestCurrency`. Each parameter has a data type and the Web service has functionality, both specified as context information. For an Interface element, we can express `hasUserID VALUE 123 and hasUserName VALUE John and hasUserAddress VALUE Dublin`. `UserAddress` and `UtilityType` are the other syntax context elements of this service. `BillRequestCurrency` is a domain context element (measures and standards). Parameter `BillRequestDevice` is a device context element, part of the platform context.

4 Constraint Integration

The ontology-based context instances, which define and describe a concrete situation, are converted into context constraints. Constraint validation services (or short constraint services as opposed to application services) validate these constraints. At composition time, context constraint validation is integrated with the application Web service process, see Fig. 2.

Each context aspect is validated by a constraint service. Constraint services use data collectors to support the validation of context constraints, e.g. when a client is using a mobile phone, the mobile phone becomes a context instance (of the device context aspect) and setting the bill format to mobile phone display becomes a required context constraint. Data collectors are used to collect device settings and constraint checkers validate the settings with the given device. Data collectors are also needed for performance constraints to determine for instance response-time behaviour. All constraints become pre- or post-conditions of the service within the integrated, composed Web service process. For response time, we determine a concrete required value for the quality attribute from the respective service profile. This is then converted into to two data collector calls (creating begin and end time stamps) and a validation constraint, which checks whether the required value is achieved. A fault exception is raised if not.

4.1 Constraint Generation

The constraint service invocations are generated based on constraint templates for the specific context aspect. A context template has link, condition type and expression elements. The link is used to support service binding. Path expressions (XPath) are used to specify the location of the constraint checker. The condition type explains both the type of the constraint validation (pre-condition or post-condition-based validation) and the order of execution. The expression specifies the constraint to be checked.

A key observation here is that the implementation of constraint validation is context category-dependent:

– The Functional context details, e.g. parameters and protocol aspects. Pre/post-condition validation is used, but no data collectors.

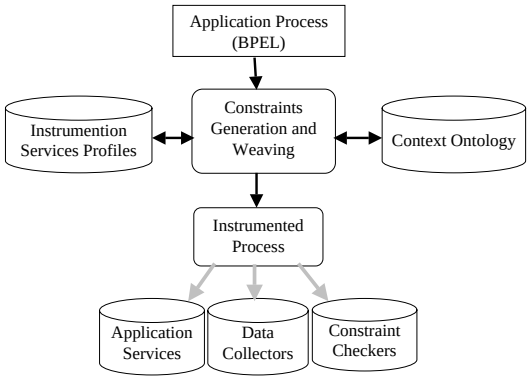

Fig. 2. Constraint generation architecture

– The Quality of Service constraints usually require data collectors to monitor variable quality properties before validating constraints.

– The Domain and Platform constraints refer to data collectors to determine environment conditions. In contrast to the quality monitors, these are static properties (such as language or device) that need to be queried, but not measured.

This category-dependency allows for uniform constraint monitoring within the categories, which is an advantage for efficient constraint integration.

4.2 Constraint Language

The Java Modelling Language (JML) is our context constraint language. JML is a behavioural interface specification language. JML is suitable as it supports pre- and post-conditions – the format in which we express validation constraints. The keyword `requires` is used in specifying pre-conditions. A pre-condition is a condition that must be satisfied before calling a service. The `ensures` keyword prefixed a post-condition that must be established. The `UserBillRequest` service proceeds further only if the user is verified (authenticated). For the `userVerification` with parameter UserID, we define:

```
<Link: path expression to the service process/>
<Condition>
  <Type> post-condition </Type>
  <Order> 1 </Order>
</Condition>
<Expression>
   @ensures returnBoolean( Context:userVerification(UserID) ) == True;
</Expression>
```

The @ensures expression requires the return value of the `userVerification` context service (located at `Context` with parameter is `UserID`) to be true. Similarly, for the `UserBillResponse` service, the constraint depends on the user device and device type (platform context):

```
<Expression>
   @ensures returnBoolean(
        Context:compareBillFormat(),
        DeviceType,
        Context:setBillFormat( Context:getBillFormat(DeviceType) ) ) == True;
</Expression>
```

This post-condition ensures that `compareBillFormat` returns true. `SetBillFormat` calls `getBillFormat`, located at `Context`, with the parameter `DeviceType` to set the bill format. The parameters are the Context-based `CompareBillFormat` service, the `DeviceType` parameter and the currently set bill format (which is set by calling `getBillFormat` in `setBillFormat`). Then `compareBillFormat` compares the device type with the set bill format and only returns true if the bill format matches the device type.

4.3 Instrumentation and Weaving

We implement dynamic constraint integration and monitoring as an instrumentation of the application process, achieved through weaving (Fig. 3) [3]. For our case study, user and service broker agree on using different devices and different currency types in the utility bill process. The user request generates both an abstract business processes and a context constraints list which is based on the agreements made between the parties. The constraints list contains the context instances and constraint validation service bindings.

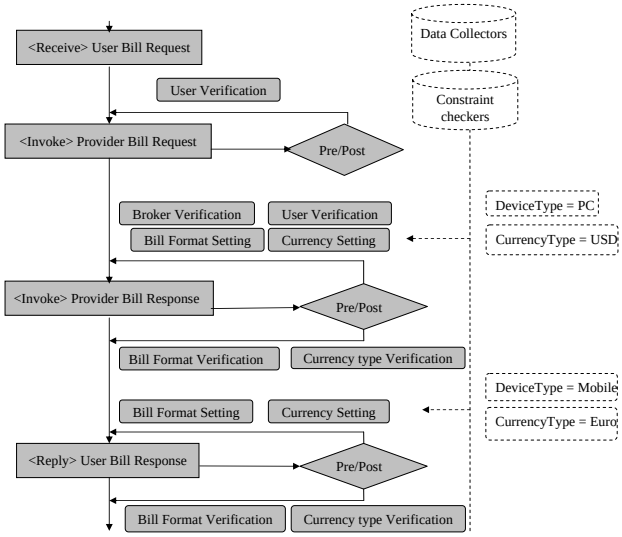

Fig. 3. Constraint validation instrumentation

At the centre of the validation instrumentation is a mapping:

– Context model attributes are connected to concrete values at instance level to form abstract constraints. Thus, attributes like `UserID` or `BillRequestDevice` are extracted from the ontology.

– A preparation step for the final mapping is the determination of data collectors (e.g. `getBillFormat`) and data initialisers (e.g. `SetBillFormat`) that support the constraint condition (e.g. `compareBillFormat`). These can be application-specific (however, `getBillFormat` might depend on a generic device context attribute). Other constraint are directly based on generic data collectors (e.g. performance monitoring services).

– The abstract constraints are mapped to JML pre- or postcondition constraints. Constraint service calls for constraint checking are generated based on context ontology instances. These service invocations are based on information given in the constraint templates, i.e. path expression (link), context constraints (condition type), constraint services and context constraint language (expression).

Constraint services encapsulate the constraint checker. A service-related context specification, the context ontology, and constraint service invocation shells can be precomputed at development time of the main services. However, the weaving process needs to be executed in parallel with web service process planner and constraints generator.

```
<bpel:invoke name="UserVerificationInvoke" ... </bpel:invoke>
<bpel:if name="userVerificationConstraint">
  <bpel:condition>
    <![CDATA[\$UserVerificationResponse.parameters/return ='true']]>
  </bpel:condition>
```

The execution of the application process only proceeds if the respective constraint is not violated. The monitoring, analysis and handling of possible violations is integrated using BPEL's fault handling mechanisms, which will be described in the subsequent sections.

5 Fault Tolerance and Remedial Strategies

Any of the context constraints might be violated at runtime. A violation causes a fault. A fault analysis takes fault data as input and is responsible for choosing a suitable remedial strategy for that fault from predefined fault remedial knowledge. The remedial strategy is applied to the faulty process execution. There are three steps for defining the fault remedial knowledge: defining a fault taxonomy, defining remedial strategies, and matching each fault category with remedial strategies.

Our fault monitoring and analysis covers application-level violations (e.g. functional of quality attributes) as well as technical, environment-specific faults (domain and platform attributes). As a consequence, our fault taxonomy is based on the context model. Thus, the root fault categories are the context categories Functional, Quality of Service, Domain, and Platform.

5.1 Defining Remedial Strategies

Some common strategies such as retry or replace have been introduced [12, 9, 2, 18]. In dynamic service composition, remedies are selected and applied dynamically. We categorise remedial strategies into goal-preserving and non-goal-preserving strategies. Goal-preserving strategies aim to recover from faults; the business goal of a process would be completed through a continued process execution after

fault recovery. Non-goal preserving strategies do not attempt recovery. They assist possible future recovery.

5.2 Goal-preserving Strategies

In BPEL, an invocation calls a business activity performed by a Web service. This can be monitored with a fault handler using the scope attachment. We identify four goal-preserving strategies for fault handling as follows:

- Ignore is a simple strategy, which does not take any action on a fault. The objective of fault-tolerance is that assuring the business goal is achieved by service processes in fault situations, rather than recovery all faults. We can ignore faults that do not affect a business goal.

- Retry is suitable for communication faults. For example, an invoked service is temporarily unavailable; messages are lost during network transmission or replies are missing. In this case, this strategy suspends the execution of the process and retries the invocation of the fault services. Maximum retry number and interval before each retry can be defined.

- Replace is similar to retry, but with an alternative service that has the same abilities as the original faulty service. Since we cannot alter remote services, replacing faulty services can be effective. However this strategy is limited to stateful Web service composition. The client is required to keep an instance or session data to support business requirements, such as conversational message exchange patterns. If the session data is non-retainable in the ongoing service, the service is tightly coupled to the process workflow. Thus, the replace strategy is unworkable.

- Recompose: Ignore, Retry and Replace are inner process-level remedial strategies which try to recover faults within the current process. Recompose is different in that it discards the fault process and re-establishes an alternative process which has same business goal. As consequence, recomposition is suitable for all categories of faults.

Ignore and Retry are lower-level recoveries that keep the original process workflow. Applying them requires less time resources. In higher-level recovery (Replace, Recompose), an additional component is needed for discovering alternatives, which requires more time and computation resources. Lower-level goal preserving strategies should be applied first, as they require less time resources with less impact on processes. The following example allows one Retry opportunity before applying Ignore:

```
<sequence>
   <Retry><max>1</max><waitingTime>P0Y0M0DT0H0M1.0S</waitingTime></Retry>
   <Ignore> <value>true</value> <log>level_1</log></Ignore>
</sequence>
```

There are two ways to provide alternative replacements. Firstly, alternative services are pre-assigned to remedial strategies. Replace can be applied instantly. Secondly, alternative services are dynamically discovered based on functional and non-functional properties. Recompose is different, as in dynamic composition, we presume service processes are only discovered at runtime. However, depending on business goal and size of the registry, Recompose can be time consuming. Hence, we have also developed a selective process repository to minimize time [17]. The process repository saves

composed services and processes with a categorized fault ratio. Alternative processes can be retrieved and selected from the process repository.

Replace is a passive technique; the backup is only called after a primary service fault. [9] introduces a parallel strategy. Several alternative services are invoked in parallel for one invocation. The first response received is chosen for ongoing process execution. A disadvantage in dynamic composition is that all alternative services need to be discovered dynamically at composition time. It also causes overheads on computation and network resources to execute alternative services. Moreover, it could cause business goal violations on state update, e.g. a bill is paid twice. The advantage is that only the best performing service is picked, and does not need to be replaced. We get similar results and avoid some disadvantages by selecting alternative services for replacement dynamically. Alternative services' fault ratio and response times in a process repository are used to determine the most suitable one.

Replace and Recompose might call for compensation or rollback. Compensation would be a pre-condition of these remedial strategies in many cases. Deploying an alternative process, the system needs to clear up partially executed faulty processes (rollback), i.e. the process execution needs transactional behaviour. However, this is difficult as no common protocol exists for Web services [16, 4]. BPEL's compensationHandler enables to define an activity at the scope or process level whose execution reverses some previously executed application logic. However, there is no automatic restoration of data during compensation. The application might define its own compensation behaviour. We assume for state-updating services that there is at least one service that can rollback its effect and does not depend on any state for execution. For Replace, compensation may also be required for post-condition faults before an alternative service is retried.

5.3 Non-goal Preserving Strategies

Non-goal preserving strategies do not impact on process execution. They can be combined with other strategies including goal-preserving strategies. We define three non-goal preserving strategies. **Log** records the captured fault. It could be applied at different levels, e.g. Level-1 logs fault source and fault message. Level-2 logs data transmission of fault sources as well. This data is saved in a fault log database. **Alert** notifies relevant stakeholders. **Suspend** suspends the faulty service or process until future investigation, if the fault element exceeds an acceptable fault ratio. The purpose is to isolate the fault elements to avoid possible repeat faults.

5.4 Fault Categories and Remedial Strategies

Matching fault categories with remedial strategies needs to consider different levels of data. From low to high, there are default remedial data, services and process-specific remedial data and application-specific remedial data.

Default remedial data comes from an analysis of fault categories. It is the proposed solution for all fault categories (see Table below). Retry is suitable for most remote faults from remote services where post-condition constraints are violated. For instance, a missingOutput fault might result from a temporary unavailable service. Retry is not suitable for pre-condition constraint violations. Replace and Recompose are suitable for all fault categories. Recompose would be last option as it is the most resource consuming.

	Pre-cond constraint violation	Post-cond constraint violation
Ignore	All fault categories	All fault categories
Retry	Not suitable	Functional context fault; Platform context fault
Replace	All fault categories	All fault categories
Recompose	All fault categories	All fault categories

The following XML code is a pre-condition remedial strategy for securityFaults. The system needs to assign a remedial strategy for each context aspect that is validated at runtime. The strategy definition is used when a fault in the respect category has been identified by the constraint service.

```
<securityFault>
   <preConditionViolationRemedy>
      <sequence>
         <Ignore><value>false</value><Ignore>
         <Retry><max>0</max><waitingTime>P0Y0M0DT0H0M0.0S</waitingTime></Retry>
         <Replace><value>any</value></Replace>
         <Recompose><value>true</value><log>level_1</log></Recompose>
      </sequence>
   </preConditionViolationRemedy>
   <postConditionViolationRemedy>...</postConditionViolationRemedy>
</securityFault>
```

Service-specific remedial data is defined according to service descriptions for specific services only. State-updating services need compensation. Services can have fault and compensation handlers associated to it.

```
<service>
   <serviceReference>
      <endpointUrl>http://localhost:8080/.../BankPaymentService</endpointUrl>
      <operation>BankPayment</operation>
   </serviceReference>
   <faults>
      <securityFault>...</securityFault>
      ...
   </faults>
   <compensation>
      <serviceReference>
         <endpointUrl>http://localhost:8080/.../BankRefundService</endpointUrl>
         <operation>BankRefund</operation>
      </serviceReference>
   </compensation>
</service>
```

Process-specific remedial data is defined according to business goals and application domains. It needs to comply with application requirements and organisational policies. In processes involving financially sensitive data, security-level mismatch faults are not acceptable; some processes would mark minor security faults as ignorable. Organisations might define their own trusted alternative service as a Replace remedy.

```
<process>
   <processReference>
      <onDemandRequest>GasBillPayment</onDemandRequest>
   </processReference>
   <services>
      <service>
         <serviceReference>...</serviceReference>
         <faults>...</faults>
      </service>
      ...
    </services>
    <faults>...</faults>
</process>
```

For a fault instance, the system searches for remedial strategies from high to low level. Higher levels are customizations of lower level data.

6 Architecture and Core Components

We divide our architecture into three layers (Fig. 4): process execution layer, composition and fault-tolerance layer and database layer. A BPEL engine is responsible for the process execution layer. The three databases of the database layer have been discussed in the previous section. The four composition components in the fault-tolerance layer form the architectural core. They directly interact with the instrumented BPEL process, thus our approach is BPEL engine independent. We discuss the core components now.

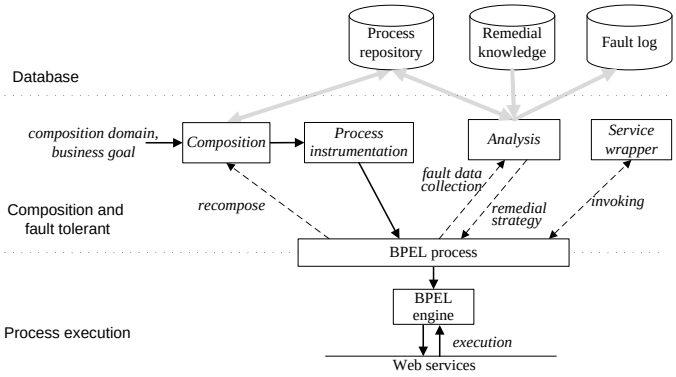

Fig. 4. Three layered system architecture

The **composition component** composes services to service processes based on user requirements – a classical AI planner can be used for this purpose [13, 31]. This requires a semantic service description to define the composition domain. Service processes are saved in an indexed process repository for possible future reuse, e.g. during recomposition. The repository supplies a suspended-list of services and processes for composition to filter invalid processes. It also contains a list of suitable replacement services for each application service (dynamic discovery and matching is beyond the scope of this investigation). To enable recomposition, the composition component is exposed as a Web service `recompose` with a `processReference` as input. `ProcessReference` contains the process

name as business goal and the process index which differentiates multiple processes for the same business goal.

The **process instrumentation component** converts application processes to instrumented processes, which includes the fault monitoring and handling mechanisms within the BPEL process.

The **analysis component** utilizes remedial knowledge to provide remedial strategies for a fault instance. It also updates the fault ratio of services and processes in the process repository and updates the fault log if the Log strategy is required. All non-goal preserving remedial strategies can be implemented by the analysis component. Its Web service interface `analyse` has five inputs. `faultData` is a fault variable or constraint violation collected by the BPEL fault handlers. `processReference` denotes the current BPEL process. `invokingServiceReference` is an instance of `ServiceReference` the identifies a Web service. A `ServiceReference` contains an endpointUrl and an invoking operation. `RequestData` and `responseData` record fault service data transmission for the Log strategy.

The **service wrapper component** is a dynamic service invoker. It wraps actual application services into a unified service interface `genericOperation`. The `genericOperation` has two input parts. `requestData` is input of the service; `invokingServcieReference` is the identity of the service. `responseData` returned by `genericOperation` is output of the service. The purpose is to provide a dynamic binding partner link. In BPEL, partner links define how a process interacts with other processes and services. Dynamic binding partner links allow that application service endpoints are selected and assigned to the partner link through configuration or runtime input. The limitation is these services must have the same interface. Our wrapper component achieves dynamic binding without this limitation.

7 Instrumentation Template for Violation Handling

The purpose of the process instrumentation is to add fault monitoring and handling capability to BPEL business processes, i.e. dynamically selected remedies are able to act on process executions when faults occur. The instrumentation is based on a designed instrumentation template for violations and fault handling for each service-invoking activity (Fig. 5). Since we use constraint services for pre- and post-activity validation, two constraint service invocations may be bound to each invoking service.

There are two types of fault handler activities in a fault handler construct: a set of custom catch activities and at most one at catchAll. A catch fault handler, which only catches a specified fault which has an optional fault name and/or fault variable. The fault data can be forwarded to the analysis component as needed. A catchAll fault handler executes if a fault is not caught by existing catch fault handlers. The catchAll ensures that no fault is ignored.

We describe the violation and fault handling template in two parts. The first part in Section 7.1 is the <`repeatUntil`> container in the top half of Fig. 5. It supports the Ignore, Retry, and Replace remedial strategies. The second part in Section 7.2 is for the Recompose strategy; see bottom half of Fig. 5. Since non-goal preserving strategies do not impact the processes, they will not be considered by the violation handling template. Again, the bill payment service process acts as an example to illustrate the application of the template to instrument the application process with fault violation handling. For instance, `billPay` is a sample service invocation that is used. Pre- and post-condition violation handling is described.

7.1 Ignore, Retry, and Replace

Five variables for each invoking activity define the violation handling context and determine the handler execution: `invokingServiceReference` provides the current invoking activity service reference, e.g. `billPay`. Variable `composition` denotes if compensation of the current invocation is needed for the Recompose strategy. It has the default value true. `compensationServiceReference` names the compensation service of the current invocation – the initial value is empty. `waitingTime` defines the waiting duration for the Retry strategy with an initial value 0. The execution `path` is by default initialised as the uninstrumented original execution path: pre-condition constraint service, invoking service, and post-condition constraint service.

In addition to context constraints, paths are the second key concept in the template. We create a conditional service composition with all possible paths, which for decisions of the analysis component include the necessary recovery as well. In a fault-free scenario, only the default path is executed. The other paths will only be executed based on a corresponding selection of `analyse`. Otherwise, the fault is caught by attached fault handlers and the `analyse` service inside the handlers determines the following actions, including selection of a new execution path. In the template, the <`repeatUntil`> container is important. It only ends when a path is executed successfully (`path=0`) or `analyse` decides to recompose the current process (`path=-1`). In the following, we distinguish pre- and post-condition based constraint violations.

For faults caused by **pre-condition constraint violations**, the fault handler passes the fault variable thrown by a constraint service (`constraintViolation`) to `analyse`. `analyse` uses `processReference`, `invokingServiceReference` and other additional variables.

1. If the remedial knowledge suggests Ignore, `analyse` returns `path = 3, compensation = true, compensationServiceReference = empty` and `waitingTime = 0` and keeps `invokingServiceReference = billPay`. The <`repeatUntil`> forces the process to execute path 3. The `billPay` is executed through the wrapper component `genericOperation` and the post-condition is validated.

2. If the Replace strategy is applied, `analyse` sets `path = 2`, `compensation = true, compensationServiceReference = empty, waitingTime = 0` and assigns the `invokingServiceReference` to an alternative service. The second path is similar to the first, except `genericOperation` replaces the original application service `billPay`. This allows the alternative service to be executed through the wrapper component.

3. If `analyse` suggests recomposition, it keeps `invokingServiceReference`, sets `compensation = true`, `compensationServiceReference = empty`, `waitingTime = 0`, and `path = -1` to end <`repeatUntil`>. We discuss this in the next subsection.

For faults coming from the **invoking service activity**, i.e. either `billPay` or `genericOperation`, non-constraint faults are caught by the catchAll handler. The fault handler assigns `path=4`, which means the post-condition constraint service deals with the fault and throws a constraint violation fault for `analyse`, i.e. a syntax constraint violation is expected thrown from the constraint service for `faultData`. Variable `compensation` is also set to false, as no compensation is required for this invocation during recomposition.

For faults caused by **post-condition constraint violation** we distinguish the four strategy cases:

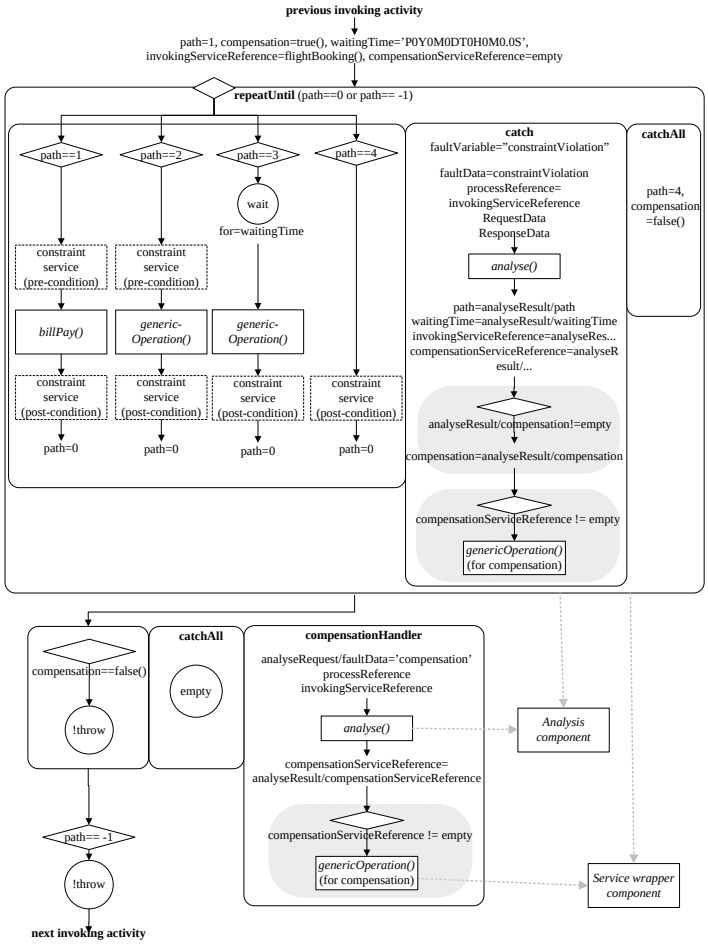

Fig. 5. Violation and fault handling template

1. For Ignore, `analyse` keeps `invokingServiceReference`, sets `compensation = empty`, `compensationServiceReference = empty, waitingTime = 0`, and `path = 0` to end <`repeatUntil`>. An empty compensation variable means to keep its previous value, i.e. the fault comes from the invoking service.

2. For the Retry strategy, `analyse` sets `path = 3, compensation = true`, `compensationServiceReference = empty` and keeps the last `invokingServiceReference`.

3. For Replace, `analyse` sets `path = 2`, `compensation =true`, `waitingTime = 0` and an alternative service for `invokingServiceReference`. Replace for post-condition faults also needs to check if compensation is required. If `analyse` returns a `compensationServiceReference`, `genericOperation` within the fault handler executes the compensation service.

4. For Recompose, `path = -1`, `compensation = empty, compensationServiceReference = empty, waitingTime = 0` is set and `invokingServiceReference` is kept.

In case of faults with alternative replacement services, the same strategy as above is applied again.

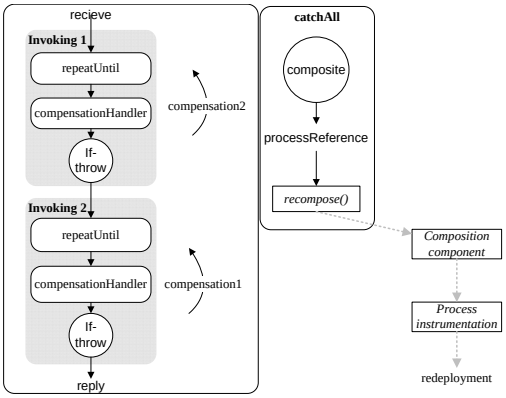

Fig. 6. The structure of a process scope

7.2 Recomposition

We continue with the second `scope` (with `compensationHandler` attachment) of Fig. 5. The second scope is responsible for compensating of an entire process, i.e. Recompose is applied. If `compensation==false`, a defined fault is thrown. An attached <`catchAll`> handler catches the fault and does nothing. The purpose is to mark this scope as faulty. The BPEL compensationHandler can only be triggered by a successful scope for process compensation. In that case, such as Ignore with a post-condition fault, a compensation handler attached to the scope within <`repeatUnit`> will never be triggered, as a fault occurred. We create this scope for invocation activity compensation, and a `compensation` variable decides whether to trigger it.

If `analyse` decides to recompose (`path=-1`), a fault is thrown. The process scope catches this fault and starts scope composition before calling `recompose` (Fig. 6). All fault-free compensation scopes are executed in backward order. If any invocation activity requires compensation, `analyse` provides a rollback service, which is executed through `genericOperation`.

8 Evaluation

We have implemented a service process monitoring architecture. Prototypes for the constraint generation and the fault handling (the two main components) exist and are the basis of the evaluation. The BPEL engine (ActiveBPEL) running the process integrates the constraint validation checker (JML checker).

Performance is central for runtime composition and monitoring. Two aspects emerge that we have investigated using our prototype:

– The constraint processing and weaving is time-consuming. Our strategy favours early pre-computation of constraint templates as instrumentation profiles (from the ontology as soon as changes to the ontology are known if the application services are determined).

– Our experiments with different variants of the architecture in terms of the constraint weaving into the BPEL service process – with respect to how constraint violations are handled – shows an acceptable overhead.

Process instrumentation makes the instrumented process more complex than the original one. For constraint violation handling, the total overhead depends on the plan execution time, the number of constraint violations, and the constraint engine performance, etc. In the case study scenarios we investigated, there is only a violation handling overhead of in average 10.05% in a total plan execution time of in average 17110 milliseconds (not including replanning). In case no faults occur, a central benefit of our solution is that there is no overhead. Since all Web services were hosted locally, we expect overheads in networked environments to be much lower.

With in total 35 test cases executed and successful, our approach provides a reliable violation and fault handling for dynamic service composition. Scalability is another central issue. In general, more complex service processes do not impact the performance and reliability results as long as the degree of concurrency does not increase significantly.

However, the overall success also depends on alternative services and processes being supplied for the replacement remedies. Dynamic replanning is another aspect that can be affected negatively with increased complexity (which is not part of this investigation). We discuss this further in the next section.

9 Discussion – Related Work, Trends and Challenges

Related work in this area covers context modelling and constraints used in dynamic service architectures. Broens proposes a context binding infrastructure called the Context-Aware Component Infrastructure (CACI) [5]. This realizes context-binding transparency and is composed of a context binding mechanism and a context discovery interoperability mechanism. However, this approach is not specific to service process composition with its binding and fault handling mechanisms. Hong and Cho present a context-aware manager architecture to support user-centric ubiquitous learning services and describe an ontology-based context model for intelligent school spaces [11], however, do not address constraint integration in service compositions. Medjahed et al. propose a context-based matching approach for Web service composition [14]. This approach introduces policy-based context binding, but does not cover violations and fault handling.

The METEOR-S project focuses on constraint-driven Web services composition [1]. It distinguishes data, functional and quality of service semantics. The use of SWRL and OWL to provide descriptive rules for specifying constraints is planned, but not realised yet. Chen et al. propose a semantic matchmaker architecture that consists of a service planner for capability matching and a context reasoner for context matching [6].

In [31, 28] solutions using various planning techniques are provided – which have received significant attention – for dynamic service composition, but they lack fault-tolerance. In [4], a constraint language is proposed for the Dynamo monitoring platform. We use a more simple and more efficient standard BPEL fault handling without requiring additional execution monitoring subsystems.

The solution that we have implemented demonstrates that fault handling mechanisms can support high-performance constraint violation handling based on standard BPEL engines. While constraint integration solves some problems, some challenges have also arisen. To provide flexibility in application processes, various types of constraints are required. Constraints need to be defined at a context model level to capture business and technical aspects and need to be integrated dynamically. Our prototype is able to integrate context constraints into application services and weave these into BPEL processes for violation handling using the instrumentation template.

Our infrastructure can support context-dependent fault handling in a composition approach. Other challenges, however, remain. A composition planner often has incomplete information initially [13]. Planners need observers or gather information. As a solution to this problem, an interleaved approach integrates service execution with sensing services as part of a planning and composition process. The composition tool dynamically queries the environment for context conditions rather than searching for all possibilities in a tree-like conditional plan. Our solution can support this through data collectors and constraint checkers. Semantic enhancements of WS-BPEL would also bring closer composition and the process execution [20].

A signalled fault does not unambiguously identify the reason. While in the context of our framework it is sufficient if a suitable remedy can be put in place (like a replacement service), the possibility to optimise the remedial strategy arises. Our aim was only the immediate dynamic reaction. An offline analysis could probe deeper in order to improve the remedial strategy definition.

10 Conclusions

Context-based composition is an essential ingredient for autonomic Web service composition. We have introduced techniques for contractual constraint violation and runtime fault handling for dynamic service composition.

An ontology-based context constraint definition and validation framework is the conceptual core. Constraint validation is woven into application service processes. We have defined the notion of context for Web services and formalized an approach to specify semantic context information in a comprehensive context ontology. We have illustrated an intelligent remedial knowledge base for dynamic remedial strategy selection. We have provided a BPEL constraint violation and runtime fault handling template to enable a lightweight engine-independent implementation. We have also evaluated our implementation in terms of violation and fault handling ability and overhead and performance aspects on instrumented process execution.

These techniques are stepping stones towards implementing a general approach for autonomic services composition. On-demand service provision is an example of changing approaches to software provision and utilisation. These new forms, however, depend heavily on dependable dynamic composition and execution.

Acknowledgment

The authors would like to thank the Science Foundation Ireland for its support of this work through the RFP project CASCAR.

References

1. R. Aggarwal, K. Verma, J. Miller, and W. Milnor. Constraint driven web service composition in meteor-s. In *Proceedings of the 2004 IEEE International Conference on Services Computing*, 2004.

2. D. Ardagna, C. Cappiello, M.G. Fugini, E. Mussi, B. Pernici, and P. Plebani. Faults and recovery actions for self-healing web services. In *15th international World Wide Web conference*, 2006.
3. L. Baresi and S. Guinea. *Towards Dynamic Monitoring of WS-BPEL Processes*, volume 3826 of *Lecture Notes in Computer Science*, pages 269–282. Springer Berlin/Heidelberg, 2005.
4. L. Baresi and S. Guinea. A dynamic and reactive approach to the supervision of bpel processes. In *1st India Software Eng. Conf.*, 2008.
5. T.H.F Broens. *Dynamic context bindings - Infrastructural support for context-aware applications*. PhD thesis, Univ. of Twente, 2008.
6. I.Y.L. Chen, S.J.H. Yang, and J. Zhang. Ubiquitous provision of context aware web services. In *IEEE International Conference on Services Computing (SCC'06)*, 2006.
7. The WS-BPEL Coalition. Ws-bpel business process execution language for web services – specification version 1.1. In *http://www-106.ibm.com/developerworks/webservices/library/ws-bpel*, 2004.
8. The World-Wide Web Consortium. Owl web ontology language. In *http://www.w3.org/TR/owl-features/*, 2004.
9. G. Dobson. Using ws-bpel to implement software fault tolerance for web services. In *32nd EUROMICRO Conference on Software Engineering and Advanced Applications*, 2006.
10. C. Doulkeridis, N. Loutas, and M. Vazirgiannis. A system architecture for context-aware service discovery. *Electronic Notes in Theoretical Computer Science*, 146:101–116, January 2006.
11. M. Hong and D. Cho. Ontology context model for context-aware learning service in ubiquitous learning environments. *International Journal of Computers*, 2, July 2008.
12. J. Lau, L.C. Lung, J.d.S. Fraga, and G.S. Veronese. Designing fault tolerant web services using bpel. In *7th IEEE/ACIS International Conference on Computer and Information Science*, 2008.
13. S. McIlraith and T.C. Son. Adapting golog for composition of semantic web services. In *8th International Conference on Principles of Knowledge Representation and Reasoning*, 2002.
14. B. Medjahed and Y. Atif. Context-based matching for web service composition. *Distributed and Parallel Databases*, 21:5–37, 2007.
15. B. Medjahed and A. Bouguettaya. A dynamic foundational architecture for semantic web services. *Distrib. Parallel Databases*, 17:179–206, 2005.
16. T. Mikalsen, S. Tai, and I. Rouvellou. Transactional attitudes: Reliable composition of autonomous web services. In *Workshop on Dependable Middleware-based Systems*, 2002.
17. C. Moore, M.X. Wang, and C. Pahl. An architecture for autonomic web servive process planning. In *3rd Workshop on Emerging Web Services Technology*, 2008.
18. O. Moser, F. Rosenberg, and S. Dustdar. Non-intrusive monitoring and service adaptation for ws-bpel. In *17th World Wide Web Conf.*, 2008.
19. M. Mrissa, C. Ghedira, D. Benslimane, and Z. Maamar. *Context and Semantic Composition of Web Services*, volume 4080 of *Lecture Notes in Computer Science*, pages 266–275. Springer, 2006.
20. J. Nitzsche, T. van Lessen, D. Karastoyanova, and F. Leymann. Bpel for semantic web services (bpel4sws). In *On the Move to Meaningful Internet Systems 2007: OTM 2007 Workshops*, pages 179–188, 2007.
21. J. O'Sullivan, D. Edmond, and H. M. Arthur. Formal description of non-functional service properties. Technical report, Queensland University of Technology, Centre for Information Technology Innovation, 2005.
22. C. Pahl and Y. Zhu. A Semantical Framework for the Orchestration and Choreography of Web Services. In *Proceedings of the International Workshop on Web Languages and Formal Methods (WLFM 2005)*. Electronic Notes in Theoretical Computer Science 151(2), 3-18. 2006.
23. C. Pahl. A Conceptual Architecture for Semantic Web Services Development and Deployment. *International Journal of Web and Grid Services*, 1(3/4), 287-304. 2005.
24. C. Pahl. A Formal Composition and Interaction Model for a Web Component Platform. In *Proceedings ICALP'2002 Workshop on Formal Methods and Component Interaction*. Electronic Notes on Computer Science ENTCS, 66(4). 2002.
25. C. Pahl, S. Giesecke and W. Hasselbring. Ontology-based Modelling of Architectural Styles. *Information and Software Technology*. 1(12), 1739-1749. 2009.
26. C. Pahl. (2005). Layered Ontological Modelling for Web Service-oriented Model-Driven Architecture. In *Proceedings European Conference on Model-Driven Architecture - Foundations and Applications ECMDA'2005* (pp. 88-102). Springer-Verlag, LNCS 3748.
27. C. Pahl. An ontology for software component matching. *Int. J. Softw. Tools Technol. Transf.*, 9(2):169–178, 2007.
28. M. Pistore, F. Barbon, P. Bertoli, D. Shaparau, and P. Traverso. Planning and monitoring web service composition. In *Workshop on Planning and Scheduling for Web and Grid Services*, 2004.

29. M.X. Wang, K. Yapa Bandara, and C. Pahl. Integrated Constraint Violation Handling for Dynamic Service Composition. In *Proceedings IEEE International Conference on Services Computing SCC 2009.* pages 168-175. 2009.
30. X.H. Wang, D.Q. Zhang, T. Gu, and H.K. Pung. Ontology based context modeling and reasoning using owl. In *Proceedings of the Second IEEE Annual Conference on Pervasive Computing and Communications Workshops.* IEEE, 2004.
31. D. Wu, E. Sirin, J. Hendler, and D. Nau. Automatic web services composition using shop2. *Workshop on Planning for Web Services*, 2003.