International Journal of Software Engineering and Knowledge Engineering
© World Scientific Publishing Company

AN ONTOLOGICAL FRAMEWORK FOR WEB SERVICE PROCESSES

CLAUS PAHL and RONAN BARRETT
Dublin City University, School of Computing
Dublin 9, Ireland
[cpahl|rbarrett]@computing.dcu.ie
http://www.computing.dcu.ie/∼[cpahl|rbarrett]

Received (Day Month Year)
Revised (Day Month Year)
Accepted (Day Month Year)

The process notion is central in computing. Business processes and workflow processes are essential elements of software systems implementations. Processes are connected to notions of interaction and composition. The Web Services Framework as a development and deployment platform for services is based on the assembly of interacting processes as the compositional paradigm. Service-based software development on and for the Web platform embracing the philosophy of discovering and using third-party services makes a shared knowledge representation framework necessary. We develop a semantical and ontological framework for service process composition. We propose a framework for the compositional definition of Web services based on the π-calculus to define protocol-like restrictions on service interactions and based on description logic and ontologies to guide the discovery and modelling of services and processes.

Keywords: Web services, service orchestration and choreography, service composition, service processes, composition ontology.

1. Introduction

The process notion is widely used in computing – in both a technical as well as application-oriented form. Processes as interacting agents has been a technical principle of organising and modelling software systems. Business processes and workflow processes are examples of the process concept in application contexts.

Recently, the Web Services Framework (WSF) has emerged as a platform for the development and deployment of service-oriented software architectures [1,3]. The assembly of interacting services to processes (implementing business or workflow processes) is the principle of architectural composition. Service-based software development on and for the WSF – the development perspective rather than deployment – receives currently increased attention. Adequate languages and techniques are needed to support a software developer the development activities. Service composition to processes is a central activity in this context [4,5,6,7,8]. The current discussion about orchestration and choreography as two forms of process composition highlights this development [9]. A comprehensive formal framework to support compositional service development is, however, still lacking.

A central requirement for the development of service-oriented architectures and processes in these architectures *on* the Web is shared knowledge representation due to the distributed and shared nature of the Web. Embracing the Semantic Web paradigm of providing and sharing semantical information is the key to the solution of the development problem. Ontologies can provide the technical infrastructure for this endeavour.

Our objective here is to develop a semantical and ontological framework for processes for the Web services platform. We propose a framework for the compositional definition of Web services based on the π-calculus to define protocol-like restrictions on service interactions and based on description logic and ontologies to guide the discovery of services and processes.

- Our aim is to give semantics to a process composition framework – semantics are central for both the internal (formalisation and definition) and external perspective (knowledge sharing),
- We will develop an ontological framework for service process description, modelling, and discovery that supports reusability and maintainability of software, i.e. to allow services to be composed and service processes to be reused.

The objective is an ontological framework that can be used as a description format and that can support essential development activities such as composition. While individual choreography and orchestration languages are defined in sufficient detail, integrated approaches based on choreography and orchestration are so far informal, using examples to motivate and illustrate differences. Ontologies for semantic Web service development [22,23] have focussed so far on the description and matching of individual services.

Our approach is based on a technical model, that captures essential semantical requirements and formally defines the platform (Section 3). Then, we address the ontological representation (Section 4). A fully formalised framework cannot be presented within the given scope here; we however discuss the central concepts behind such a formalisation. Here we integrate and apply to Web service composition a number of results that we have presented elsewhere [14,27,28]. Since semantic Web services have been widely addressed, we also show how our framework can be integrated with semantic Web services (Section 5). A broader range of applications and tool implementations based on the technical and ontological framework and a comprehensive discussion follows (Section 6). We start, however, with an outline of service process development in the next section.

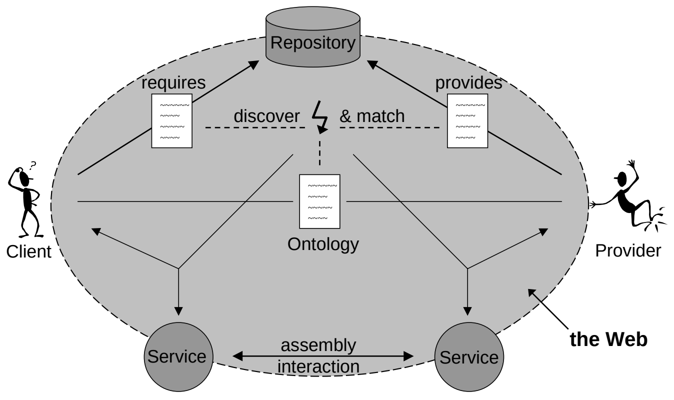

Fig. 1. The Web as a Service-oriented Development and Deployment Platform.

2. Development of Composite Service Processes

The term *process modelling* is associated with the dynamic behaviour of organisations, businesses, or systems. The basic idea is that these systems can be thought of as operating or behaving as a number of interrelated processes. Process models are constructed that can form the basis of software systems to support the processes. Important in our context is active modelling, i.e. considering the execution of process actions in a software infrastructure such as the Web Services Framework (WSF) platform.

A *service* is made up of a coherent set of operations provided at a certain location. The service provider makes an abstract service interface description available that can be used by potential service users to locate and invoke the service. Services are often used 'as is' in single request-response interactions [10], but more and more the *composition of services* to *processes* is important – the recent discussion about orchestration and choreography of Web services supports this observation [9]. Existing services can be used to form business or workflow processes. The *principle of architectural composition* here is *process assembly*.

The WSF provides a platform to invoke services on a once-off basis. Real value, however, will be added if services can be composed [9]. Orchestration and choreography are two forms of service composition and collaboration.

- *Orchestration* refers to a composed business process that may use both internal and external Web services to fulfil its task. The business process is controlled by one of the agents in the system. The process is described at the message level, i.e. in terms of message exchanges, focusing on the execution order.
- *Choreography* addresses the interactions that implement the collaboration between services. Multiple agents are considered. Either each agent de-

scribes its own part in the interaction or a global perspective is taken focusing on the connections.

The discovery and invocation infrastructure – a registry or marketplace where potential clients can search for suitable services and an invocation protocol – with the services and their clients form a *service-oriented architecture*. Protocols and languages for description and composition are central elements of a service-oriented architecture. Fig. 1 illustrates this infrastructure for the WSF with a repository that can implement the registry or marketplace, a service interaction infrastructure and ontologies are central knowledge bases that integrate the various sofware engineering activities for service development and deployment. Software engineering for service-oriented architectures is a two-step process – assuming that service repositories are available:

- *Discovery* is based on abstract computation descriptions (and other software properties), formalised based on ontologies – see upper half of Fig. 1.
- *Assembly* and *usage* is about composition of matching service processes and their interaction through invocation – see lower half of Fig. 1.

An essential part of service-orientation is the possibility to make a service available to other users. This requires adequate semantical, ontology-based description.

A *sample application* – an *online shopping system* – shall accompany our service process framework. In Fig. 2, five services with their interfaces and interaction processes are described. It is the latter processes that we are interested in. Each of these services implements a process – the orchestration perspective. `Login;((Catalog+Quote)*;Purchase)*;Logout` is a process expression describing an interaction process of an online shopping user starting with a login, then repeatedly buying products (which consists of an internal loop of product retrieval – catalog browsing or quotation enquiries – and then purchasing), before logging out.

The interactions resulting from the service invocations (e.g. `Catalog!` as output) and service provision (e.g. `Login?` as input) are the result of service choreography, see Fig. 3. For instance, `ShoppingProcess` is a client of `CatalogServer`, `PurchaseServer` and `LoginServer`; `PurchaseServer` is a client of `Stocks`.

As part of a development, the shopping process needs to be implemented. The `ShoppingProcess` is linked to three other services, i.e. `CatalogServer`, `PurchaseServer`, and `LoginServer`, through three different connectors. For instance, the subprocess `(Catalog?+Quote?)*` with the corresponding operation signatures forms a requirements specification that has to be matched by an existing provided service – here the `CatalogServer` satisfies the requirements. The term `(Catalog+Quote)*` is only part of the full shopping behaviour and relates only to the first connector. The purchase-part is dealt with by the second connector.

```
Service ShoppingProcess
  operation  import Login [no:int,user:string] : bool
             import Catalog [ID:int] : product
             import Quote [prod:product] : price
             import Purchase [prod:product] : void
             import Logout [no:int] : void
  process    Login!;((Catalog!+Quote!)*;Purchase!)*;Logout!

Service CatalogServer
  operation  export Catalog [ID:int] : product
             export Quote [prod:product] : price
  process    (Catalog?+Quote?)*

Service PurchaseServer
  operation  export Purchase [prod:product] : void
             import Available [ID:int] : bool
  process    (Purchase?;Available!)*

Service Stocks
  operation  export Available [ID:int] : bool
  process    (Available?)*

Service LoginServer
  operation  export Login [no:int,user:string] : bool
             export Logout [no:int] : void
  process    (Login?+Logout?)*
```

Fig. 2. Bank Account Services and Processes – Orchestration Aspects.

3. Services and Processes – an Operational Framework

Description and composition are central design activities. In this section, we develop an abstract language and a technical model that form an operational framework for both activities. We formalise orchestration and choreography within this semantical framework. This framework serves to capture requirements and forms an underlying layer for the ontological framework.

3.1. *Orchestration and choreography description*

Various orchestration and choreography languages have been proposed by various organisations. We follow the discussion in [9] to extract the central language concepts.

We chose the π-calculus [13] as the basis for our framework. Classical service composition models that focus on simple input/output-oriented functional behaviour are not adequate here. While choreography is often about fixed connections, in

```
Connector Catalog
  connection  ShoppingProcess SP, CatalogServer CS
  messages    Catalog [out SP, in CS] int -> prod
              Quote [out SP, in CS] product -> price

Connector Purchase
  connection  ShoppingProcess SP, PurchaseServer PS
  messages    Purchase [out SP, in PS] prod -> void

Connector Login
  connection  ShoppingProcess SP, LoginServer LS
  messages    Login [out SP, in LS] int,string -> bool
              Logout [out SP, in LS] int -> void

Connector Stocks
  connection  PurchaseServer PS, Stocks ST
  messages    Available [out PS, in ST] int -> bool
```

Fig. 3. Bank Account Services and Processes – Choreography Aspects.

order to support evolution and change management, a flexible connection management for services is required. The π-calculus is a calculus for mobile, distributed systems. Mobility in the π-calculus is achieved through the possibility of passing channel names along connections, which can be used by the recipients as references to create connections dynamically. The π-calculus is a formal calculus focusing on (bi-)simulation as a notion of process equivalence based on observable behaviour. It has two benefits over classical process algebras such as CSP or CCS. We use the similarity between mobility and evolution here – both are about changes in the relationship to other agents or services – to address flexible connection management. We also use this approach to allow service providers and clients to agree on interaction channels dynamically.

3.1.1. *Orchestration*

We can derive the following core requirements for an orchestration notation from languages such as WS-BPEL [9]:

- *basic elements*: message-based actions in two forms – invocations for external services and receive/reply actions if the service is available to others,
- *process language*: sequence, choice, iteration, and concurrency are the service process combinators,
- *abstraction and export interface*: a process can be provided as a Web service,

- *state and data*: variables and parameters for actions are needed.

A number of other, more advanced aspects can also be identified. These include transactions and exception handling. We, however, focus on the more central ones here. The focus of orchestration is visualised in Fig. 5. The business process itself and the Web services that implement the process are separated. This keeps the process logic apart from its implementation. A process description can serve different purposes:

- to define a business process in terms of actions and control flow – and also in terms of the concrete services that are used,
- to describe the external, observable interaction pattern that a service can engage in a composed system (if the process is made available as a service).

A process is executed by an orchestration engine which invokes the respective service operations for each process action.

We capture the foundations of orchestration in form of a *process language and model* focussing on aspects of service compositions. A process description is about control flow and the determination of the execution order. We start with abstract actions to concentrate on control flow first – data aspects and also interactions will be added later. **Service processes** are inductively formed based on basic process names, named process expressions, and combinators, see upper part of Fig. 4. A named process expression (an abstraction) is defined by a service process expression. The process definition is recursive. Based on basic processes (which are Web services), composite service processes can be defined, i.e. expressions such as $P = s_1; s_2; Q$ can be used. We also use the notation $P \xrightarrow{s_1;s_2} Q$ to emphasise the transitional character of processes. Note, that we often drop the parameter list [..] of actions, if we are only interested in the process control flow, i.e. use $s!$ instead of $s![x]$.

Example 1. We can specify an abstract business process for an online shopping system:

```
OnlineShopping = Login; ((Catalog + Quote)*; Purchase)*; Logout
```

This orchestration example ignores the import/export classification of process elements necessary for choreography as well as data aspects.

We can add import/export directions and data by refining the notion of actions, see lower part of Fig. 4. Receive and reply actions are needed to provide functionality in form of a service. Invocation is needed to use other services. The invocation provides a scope for the returned result y of the interaction.

3.1.2. *Choreography*

Similar to our orchestration discussion, we summarise the main requirements for a choreography description notation based on languages such as WSCI [9] and WS-CDL

Process Expression:

P	$::=$	A	action
		$P_1 ; P_2$	sequential composition
		$P_1 \| P_2$	parallel composition
		$P_1 + P_2$	non-deterministic choice
		$P*$	iteration

Abstraction:

$Q[s_1, \ldots, s_n]$	$=$	P	where $s_1, \ldots, s_n$ services in P

Actions:

$s?[x]$	receive action
$s![x]$	reply action
$let\ y = s![x]\ in\ P$	invoke action

assuming service operation s and data item x

Fig. 4. Orchestration Language.

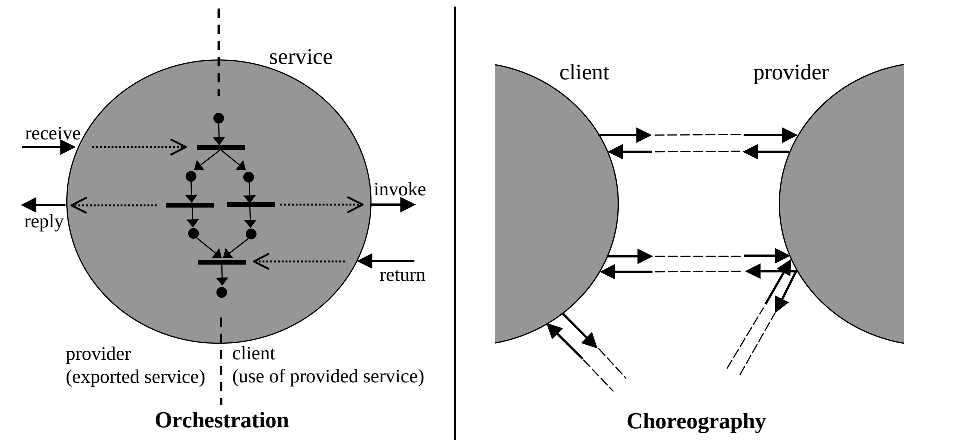

Fig. 5. Principles of Orchestration and Choreography.

^{2}:

- *basic activities*: request and response actions for local activities, invoke to call operations of external services,
- *structured activities*: loop, sequence, choice, and concurrency,
- *infrastructure*: channels (connections) between ports, which represent services and their operations.

Actions:

$s?[x]$	receive action
$s![x]$	send action

assuming service operation s and data item x.

Interaction:

$s![x]; s?[y]$	invoke-return: for each operation s in P a write-read sequence where y is the returned result from an external service
$s?[x]; s![x]$	receive-reply: for each operation s in P a read-write sequence where f is some internal service functionality

assuming a process expression P.

Fig. 6. Choreography Language.

The focus of choreography is observable interaction behaviour, not execution; see its visualisation in Fig. 5. The orchestration model is a *process model* with its focus on control flow and execution order. The choreography model is an *interaction model.* It is about the interaction of processes, i.e. synchronisation and exchange of data. Essential in modelling service process interaction is to add data and message flow between service processes.

Web services are connected through a network. The network endpoints that represent services are called *ports* – service names act as port names. Services (and their ports) can be receivers and senders of data, i.e. read from or write to communication channels set up between the ports, see Fig. 6 upper part. Note, that in contrast to orchestration, we have abstracted here from the difference between provider actions (receive/reply) and client actions (invoke). An **interaction** is the activation of a remote service. Two forms shall be provided, see Fig. 6 lower part. These interactions are the basic building blocks of the process life cycle.

Example 2. The interaction `Quote![prod];Quote?[price]` is an invoke-return example that asks a service `Quote` for a quote for product `prod` and receives the `price` in the following action using the same channel.

All input service names in a process expression need to be bound to a concrete service that can execute the service functionality. Finding suitable services that match each individual service requirements is part of the process model and its matching support, and managing the connections is part of the interaction model and its connection support.

So far, the concurrent composition of processes does not allow interactions. A transition rule (called reaction rule in the π-calculus) can capture interaction and describe the data flow in these interactions – see Fig. 8. A *shared channel* can be created that forms a *connection* between two service processes. Usually, the port names act as channel names (the π-calculus requires matching port names to establish a connection; we will loosen this constraint later on). Choreography is often about fixed connections [9], but in order to enable dynamic establishment and changes in choreographies, we use the π-calculus approach to mobility.

3.1.3. *Interpretation and semantics*

Processes are composed of individual services and their provided operations. Each of these is a state transition, i.e. transforms a state of an underlying system into another. The process expressions shall therefore be interpreted in Kripke transition systems [11], a form of labelled transition systems.

A **Kripke transition system** (KTS) is a semantic structure $\{S, L, T, I\}$ consisting of a set of states S, a set of action labels L, a transition relation T on S, and an interpretation I. Service processes P are **interpreted** in KTS as transition relations $P^I \in T$ on $S \times L \times S$:

- sequential composition executes one process after the other,
- the choice operator chooses one process non-deterministically,
- the iteration repeats the process a non-deterministically chosen (finite) number of times,
- parallel composition executes both processes.

Addressing the semantics here is important as it will allow us later to formally integrate this technical framework with the ontological framework.

3.2. *Composition support*

Descriptions are needed for providers to publish their services in accessible repositories and for potential clients to capture service requirements. Service processes, whose descriptions match, can be composed. We provide a simple development and deployment model for services in form of a life cycle model, before addressing techniques needed for individual activities such as matching and composition in that life cycle.

3.2.1. *Life cycle and activities*

Description and matching are design activities. Essential is, however, the support of the full process life cycle to integrate deployment aspects as well [12,28]. Each service s in our interaction model is actually a **family of ports** s_C, s_I, s_R that address the needs of the different life cycle activities. Port s_C is a *contract port*, representing an interface that captures abstract properties. s_I and s_R are *connector ports* for interaction – s_I handles service invocation and input and s_R handles the service reply. We express the **service life cycle** in an annotated process notation based on the three port types

$$\textsc{Req}\ s_C![s_I];\ (\ \textsc{Inv}\ s_I![a, s_R];\ \textsc{Res}\ s_R?[y]\ )*$$

for the *client* with annotations for requesting, invoking, and result. Dual to the client view, there is a *provider* view

$$\textsc{Pro}\ s_C?[s_I];\ (\ \textsc{Exe}\ s_I?[a, s_R];\ \textsc{Rep}\ s_R![f(a)]\ )*$$

with annotations for providing, executing, and replying. In the client view, $\textsc{Req}\ s_C![s_I]$ is an annotated output action of service s. A process or service can request $\textsc{Req}$ a service using contract port s_C. Connector port references s_I and s_R are subsequently sent for further interactions. If matching between a client port and a provider port is successful, then the client and the provider process can be composed, i.e. a client can interact with the provided service repeatedly. The client would invoke $\textsc{Inv}$ the service at port s_I and receive a result $\textsc{Res}$ at port s_R. Here, the π-calculus mobility approach is exploited – channel names are sent along the connections, which allows clients and providers to automatically and, if necessary, dynamically decide upon interaction channels for service execution.

Activities are captured in a standard life cycle form, which represents a *composition and interaction protocol*, see Fig. 7, that formalises the development and deployment scenario given in Fig. 1. This protocol integrates development metamodel elements such as matching and composition with concrete deployment process interactions. Clients C are parameterised by their required services. All service requests need to be satisfied – expressed by a parallel composition of individual ports – before any interaction can happen. Service providers P need to be replicated in order to deal with several clients at the same time. Providers do not need to engage in interactions with all their ports – modelled by using the choice operator instead of the parallel composition for client services. Clients C and servers S can then be composed in parallel to form a system.

A service process is often both service client and provider (e.g. `PurchaseServer` in our sample application), which would require an extension of the protocol. Requirements $\textsc{Req}\ m_C^i![m_I^i]$ $(i = 1, .., l)$ have to be satisfied and connectors have to be established, before any service $\textsc{Pro}\ n_C^j?[n_I^j]$ $(j = 1, .., k)$ can be provided.

3.2.2. *Matching*

Matching is central in composition. An existing provided service that is reused and integrated, for instance into a business process, must match the client requirements in order to allow the business process to fulfil its task. To support matching of required and provided processes is our main goal.

- *Import processes* describe how a process expects to use other services.
- *Export processes* describe how provided services have to be used.

These two process description forms are elements of an orchestrated business process. Orchestration elements are more relevant to matching than choreography aspects such as interaction, which is more deployment-oriented.

Client:

$$
\begin{aligned}
& C[m_1, \ldots, m_l] \overset{\text{def}}{=} \\
& \quad \textsc{Req}\ m^1_C![m^1_I]; (\textsc{Inv}\ m^1_I![a^1, m^1_R]; \textsc{Res}\ m^1_R?[y^1])* \\
& \ \mid \ \ldots \ \mid \\
& \quad \textsc{Req}\ m^l_C![m^l_I]; (\textsc{Inv}\ m^l_I![a^l, m^l_R]; \textsc{Res}\ m^l_R?[y^l])*
\end{aligned}
$$

Provider:

$$
\begin{aligned}
& P[n_1, \ldots, n_k] \overset{\text{def}}{=} \\
& (\quad \textsc{Pro}\ n^1_C?[n^1_I]; (\textsc{Exe}\ n^1_I?[y^1, n^1_R]; \textsc{Rep}\ n^1_R![f(y^1)])* \\
& \quad + \ \ldots \ + \\
& \quad \textsc{Pro}\ n^k_C?[n^k_I]; (\textsc{Exe}\ n^k_I?[y^k, n^k_R]; \textsc{Rep}\ n^k_R![f(y^k)])* \ )*
\end{aligned}
$$

Fig. 7. Composition and Interaction Protocol.

The specification of service processes describes the ordering of observable activities of a process. We use a notion of simulation to define process matching. The requested process is the import process pattern that the client expects the provider to support through the export process pattern.

> A provider process P **simulates** a requested client process C if there exists a binary relation $\mathcal{R}$ over the set of processes such that if whenever $C\mathcal{R}P$ and $C \xrightarrow{m} C'$ then there exists P' such that $P \xrightarrow{n} P'$ and $C'\mathcal{R}P'$. We say that P **matches** C in this case.

This definition originates from the simulation definition of the π-calculus [13]. In order to determine simulations, i.e. to decide matching, we need the notion of a transition graph. A **transition graph** $G = (N, E)$ for a composite process P and a KTS (S, L, T, I) for P is a graph that represents all possible executions of P with $N \subseteq S$ subset of states and $E \subseteq T$ subset of relations. A process simulates another if we can construct a homomorphism between the transition graphs of the process expressions. A transition graph can be constructed inductively over the syntactical structure of a composite process expression. This means that the relation can be computed. In [14], we have presented a constructive form of determining the simulation via the calculation of transition graphs.

The provider needs to be able to simulate the request, i.e. needs to meet the requirements of the client. However, this is not a bisimulation − irrelevant elements in the provider process are not permitted. Dynamic binding of concrete services to the process names is possible. Our matching definition is about *potential* interaction, and not only fixed connections as assumed in most choreography languages.

Example 3. The provider provides a service process

```
Login;(Catalog+Quote)*;Purchase
```

and the requestor expects support of the process

```
(CatalogBrowse+QuoteProd)*;ProdPurch
```

If the pairs of service operations `Catalog/CatalogBrowse`, `Quote/QuoteProd`, and `Purchase/ProdPurch` match based on their individual service descriptions (e.g. signature equality), then the provider matches (i.e. simulates) the requested process.

3.2.3. *Connection and interaction*

Composition consists of two activities: matching and connection. Successful matching can result in a connection between service ports. From the perspective of a business process, concrete services are connected to the abstract business process actions. So far, we have been looking at matching of abstract process descriptions. We now focus on the computational side of compositions. The connection of matching services shall now be formalised using an operational execution semantics.

In the composition process we can distinguish a *contract phase* where both process instances try to form a contract based on matching abstract descriptions. The *connection phase* establishes a connector channel for interaction between the services. We capture contract and connector establishment in form of two transition rules. This formalises the connection of provider and client in the WSF – a virtual link between URIs that is used by for instance the SOAP protocol.

For a parallel composition of a client $m_C![m_I];C$ and a provider $n_C?[n_I];P$, both processes commit themselves to a communication along the channel between ports m_C and n_C, if their specifications match. We define a **composition** $C \frown P$ as $\{c/m_I\}C|\{c/n_I\}P$ where c is a private channel between the two processes, i.e. it is a parallel composition where a private channel, the connector, replaces the port names for the interaction. The **contract rule**, see Fig. 8, formalises the process of matching and commitment. The arrows $\rightarrow$ denote state transitions of the individual processes, either through observable actions $x![y]$ and $x?[y]$ or through internal, non-observable *interactions* τ. The contract rule differs from the π-calculus reaction rule which requires channel names to be the same [13]. We only require equality of signatures. Type systems for the π-calculus usually constrain data that is sent; we constrain interaction between processes.

Example 4. The client requires a service (annotation REQ) through port $\texttt{Quote}_C$ and the server provides a service (annotation PRO) through port $\texttt{QuoteProd}_C$

$$\texttt{Clt} \overset{\text{def}}{=} \textrm{REQ}\ \texttt{Quote}_C\texttt{![Quote}_I\texttt{];Clt}'$$
$$\texttt{Pro} \overset{\text{def}}{=} \textrm{PRO}\ \texttt{QuoteProd}_C\texttt{?[QuoteProd}_I\texttt{];Pro}'$$

which can result in a commitment of contract ports.

A connector is created if a client requesting m_I invokes a service n_I at the server side, described by the **connector rule**, see Fig. 8. Parameter data a and a reply channel m_R are sent to the provider. Parameter a replaces x in P.

Contract Rule:

$$\frac{\textsc{Req}\ m_C![m_I];C \xrightarrow{m_C![m_I]} C \quad \textsc{Pro}\ n_C?[n_I];P \xrightarrow{n_C?[n_I]} P}{\textsc{Req}\ m_C![m_I];C+M_1 | \textsc{Pro}\ n_C?[n_I];P+M_2 \xrightarrow{\tau} C \frown P} \langle\ sign(n_C)=sign(m_C)$$

Connector Rule:

$$\frac{\textsc{Inv}\ m_I![a,m_R];C \xrightarrow{m_I![a,m_R]} C \quad \textsc{Exe}\ n_I?[x,n_R];P \xrightarrow{n_I?[x,n_R]} P}{\textsc{Inv}\ m_I![a,m_R];C+M_1 | \textsc{Exe}\ n_I?[x,n_R];P+M_2 \xrightarrow{\tau} C \frown \{a/x\}P} \langle\ sign(n_I)=sign(m_I)$$

where *sign* is a function that represents the interface signature of individual operations (input- and output parameters) and M_1 and M_2 are arbitrary processes.

Fig. 8. Contract and Connector Rule.

Example 5. The composition of `Pro'` and `Clt'` creates a connector that allows the client to use a provided service, e.g. `QuoteProd`.

$$\texttt{Clt}' \overset{\text{def}}{=} \textsc{Inv}\ \texttt{Quote}_I\texttt{![pid].Clt}''$$
$$\texttt{Pro}' \overset{\text{def}}{=} \textsc{Exe}\ \texttt{QuoteProd}_I\texttt{?[x].Pro}''$$

The requestor can invoke (Inv) a service through the interaction port $\texttt{Quote}_I$, which will trigger the execution (Exe) of $\texttt{QuoteProd}_I$ with parameter `pid` by the server.

4. Services and Processes – an Ontological Framework

Service-based software development on the Web is ideally supported through ontology technology to enable the shared representation of knowledge, here service and service process descriptions, and reasoning about this service knowledge, see Fig. 1. We illustrate what ontology technology can do for service process composition and how description, discovery, and composition of services processes can be represented in and supported by a description logic that underlies a Web ontology language.

4.1. *A knowledge space for service processes*

Before developing an ontological framework for service processes, we explore the process notion from a wider knowledge representation point of view. We define a *knowledge space* for service-based processes by identifying its main facets. *Ontologies* are knowledge representation frameworks. In our context, two *types of knowledge* are important: *domain-specific* knowledge about the context of the process deployment (which we neglect) and *software-specific* knowledge about technical aspects of services and processes (which we focus on).

In general, *knowledge representation* [15] is concerned with the description of entities in order to define and classify these. Entities can be distinguished into objects (static entities) and processes (dynamic entities). *Processes* are often described in three *aspects* or *tiers*:

	Knowledge Aspect	**Knowledge Type**	**Function**
Discovery	intention (terminology)	domain	taxonomy, thesaurus
Composition	effect (behaviour)	service/process and activities	conceptual model, logical theory
Execution	form (implementation)	service/process	conceptual model

Fig. 9. Development Activities and Knowledge Space Facets.

- *Form* – process and implementation – the 'how' of process description,
- *Effect* – abstract behaviour and results – the 'what' of process description,
- *Intention* – goal and purpose – the 'why' of process description.

We have related the aspects form, effect, and intention to software characteristics such as processes and abstract behaviour. Services are software entities that have process character or can be assembled to processes. We can use this three-tiered approach for their description.

We can distinguish four *ontology functions* [16] that characterise how knowledge is used to support the process modelling activities:

- *Taxonomy* – terminology and classification to support structuring and search. Basic taxonomies can support for instance service signatures.
- *Thesaurus* – terms and their relationships to support a shared, controlled vocabulary. Dealing with equality and equivalence is an advanced thesaurus functionality.
- *Conceptual model* – a formal model of concepts and their relationships of the application domain and the software technology context. A conceptual model for service processes is the aim here.
- *Logical theory* – logic-supported inference and proof applied to behavioural properties. Matching is the main activity here, supported by a logical theory.

Fig. 9 summarises activities for the development of service-based software systems and knowledge space aspects. It relates the activities discovery, composition, and execution on services (with the corresponding ontologies) to the three knowledge space facets.

The technical framework (see Section 3) goes beyond what we need for the ontological framework in order to support the core development activities for service-based software systems. Ontologies, which fill the knowledge space, are needed to support discovery through description and composition through matching of processes, i.e. port orientation and other interaction and choreography aspects are less relevant. The ontological framework therefore abstracts the underlying operational framework, which defines the development and deployment infrastructure.

We develop the ontological framework in terms of a *description logic* [17]. Description logic as an underlying logic of the Semantic Web is particularly interesting for the software development context due to a correspondence between description logic and dynamic logic (a modal logic of programs) [18]. This correspondence, based on a similarity between quantified constructors (expressing quantified relations between concepts) and modal constructors (expressing safety and liveness properties of programs), can add process-specific reasoning support to our framework. Dynamic logic program expressions correspond to the process expressions we introduce into description logic – effectively realising a simple dynamic logic in a description logic. This allows us to incorporate modal reasoning about programs and processes into a description logic framework.

4.2. *A basic process ontology*

Ontologies are formal frameworks that provide various functions through knowledge description and reasoning techniques. The starting point in defining an ontology is to decide what the basic ontology elements represent. Here, the ontology shall formalise process-based, i.e. state-transition-based software systems. Three elements define the ontology language: concepts, roles, and constructors.

- **Concepts** are classes of objects with the same properties. **Individuals** are named objects. Concepts represent software system properties in this context. Systems are dynamic. Descriptions of properties are inherently based on underlying notions of state and state change.
- **Roles** in general are relations between concepts. Here, they shall represent two different kinds of relations. **Transitional roles** represent service operations in form of accessibility relations on states, i.e. they represent services resulting in state changes. **Descriptional roles** represent properties of a state such as invariant descriptions like service name and description or pre- and postconditions (if they are part of the description format).
- **Constructors** allow more complex concepts to be constructed in form of **concept descriptions**. Classical constructors include conjunction $\sqcap$ and negation $\neg$. Hybrid constructors are based on a concept and a role. The constructor $\forall R.C$ is interpreted based on either an accessibility relation R to a new state C for transitional roles, or based on a property R satisfying a constraint C for descriptional roles.

Our service process ontology is presented in Fig. 10. A state is an abstract concept (represented by circles) that is described in terms of elements of auxiliary domains through descriptional roles such as mutable and invariant state properties (formal

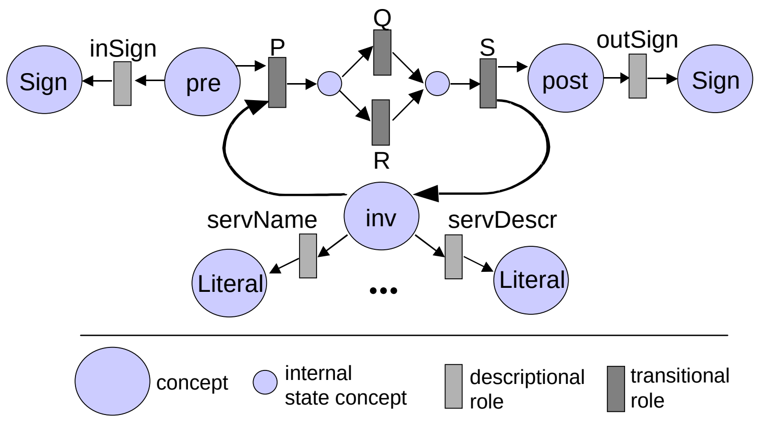

Fig. 10. Sample Service Process Ontology.

conditions, textual descriptions, etc.). The two essential state concepts, presented at the centre of the ontology visualisation, are `pre` and `post`, which denote abstract pre- and post-*states* for service process transitions (not to be confused with pre- and post*conditions*). For example, $\forall$`outSign.int` specifies a post-state by associating an output signature `int`. Roles that connect the state and description concepts are represented through rectangles.

We interpret concepts and roles in Kripke transition systems, as we did with the technical framework. This is the key to a formal integration of the process view we looked at earlier on and the ontological view we are looking at now. Kripke transition systems are used to interpret modal logics; they also suffice to interpret description logics. Concepts are interpreted as sets of states. Transitional roles are interpreted as accessibility relations. Descriptional roles are interpreted as relations involving other concept domains.

4.3. *Orchestration and choreography*

4.3.1. *Orchestration*

We have introduced the representation of basic services in a description logic-based ontology in Section 4.2. An ontology that captures service processes and their composition, however, requires an extension of classical description logics. So far, roles – representing service operations – are atomic. **Role constructors**, see Fig. 11, allow us to integrate process description and composition into an ontology framework. Note, that usually the constructors $R \circ S$, R^+, $R \cap S$, and $R \cup S$ are used, see [17]. This new language is a first extension of $\mathcal{ALC}$, a standard ontology language. So far, this extended language realises a conceptual model for the representation of service processes.

Basic Concepts:

C	$::=$	A	atomic concept
		$C_1 \sqcap C_2$	conjunction
		$C_1 \sqcup C_2$	disjunction
		$\neg C$	negation

Concept Descriptions:

C	$::=$	$\forall R.C$	value restriction
		$\exists R.C$	existential quantification

Role Constructors:

R	$::=$	A	action
		$R_1; R_2$	sequential composition – we also use $R_1 \circ R_2$ for functional composition
		$R_1 \vert R_2$	parallel composition
		$R_1 + R_2$	non-deterministic choice
		$R*$	iteration

Names and Parameters:

n_N	name n with $n_N^I = \{(n^I, n^I)\}$ for the interpretation n^I of n
$R \circ n_N$	parameterised role with transitional role R and name n_N

Fig. 11. Ontology Language.

Example 6. A description logic expression such as

$\forall$ `(Catalog+Quote)*;Purchase.postState`

that describes a composite process is now permitted.

Axioms and inference rules allow us to capture activity-related properties in the logic, for instance in order to reason about matching. The **axioms**

$$\begin{array}{rcl} \forall R.\forall S.C & \Leftrightarrow & \forall R\,; S.C \\ \forall R+S.C & \Leftrightarrow & \forall R.C \sqcup \forall S.C \end{array}$$

describe the conversion between logical operators and role expression combinators, acting somewhat like a thesaurus for service process expressions.

We integrate data and process parameters into the logic in our second language extension step. We introduce data in form of *names*, see Fig. 11. We follow an approach here that we developed originally for component composition [14] and adapted to the Web service context in [27]. Names stand for individual data elements. **Names** are denoted by a role, interpreted by an identity relation. A **parameterised role** is a transitional role applied to a name. We often drop the $_N$-postfix when it is clear from the context that a name is referred to.

Service `PurchaseServer`

operation

$$\begin{aligned}
\texttt{pre}_{Purchase} &\equiv \forall \texttt{Purchase} \circ \texttt{prod}_N.\texttt{post}_{Purchase} \\
&\sqcap \forall \texttt{inSign}.\texttt{product} \\
\texttt{post}_{Purchase} &\equiv \forall \texttt{outSign}.\texttt{void} \\
\\
\texttt{pre}_{Available} &\equiv \forall \texttt{Available} \circ \texttt{ID}_N.\texttt{post}_{Available} \\
&\sqcap \forall \texttt{inSign}.\texttt{int} \\
\texttt{post}_{Available} &\equiv \forall \texttt{outSign}.\texttt{bool}
\end{aligned}$$

process

$$\texttt{pre} \equiv \forall \texttt{!(Purchase + Available).post}$$

invariants

$$\begin{aligned}
\texttt{inv}_{Purchase} &\equiv \forall \texttt{opName}.\{\texttt{"purchase"}\} \\
&\sqcap \forall \texttt{opDescr}.\{\texttt{"executes purchase of product"}\} \\
&\sqcap \forall \texttt{Purchase} \circ \texttt{prod}_N.\texttt{inv}_{Purchase} \\
\texttt{inv}_{Available} &\equiv \ldots
\end{aligned}$$

Fig. 12. Ontological Specification of the `PurchaseServer`.

Example 7. Given a transitional role `Login` and a descriptional role `outSign`

$$\forall\ \texttt{Login} \circ \texttt{[id}_N\texttt{,pwd}_N\texttt{]}.\forall \texttt{outSign}.\,\texttt{bool}$$

means that by executing $\texttt{Login} \circ \texttt{[id}_N\texttt{,pwd}_N\texttt{]}$ a state can be reached that is described by a result value with signature `bool` . The term $\texttt{Login} \circ \texttt{[id}_N\texttt{,pwd}_N\texttt{]}$ is a composite role expression in which the identifiers $\texttt{id}_N$ and $\texttt{pwd}_N$ are constant roles (names).

With the ontology language, Fig. 11, now being complete, we can look at a more complete example.

Example 8. In Fig. 12, we have specified the `PurchaseServer` from Fig. 2 in an ontological notation.

The process ontology we have developed here is closely related to the process-based orchestration language – compare Figs. 4 and 11. We have already discussed the semantical integration through Kripke transition systems earlier on. We demonstrate how other constructs such as matching can also be represented in the logic in the remainder of this section.

4.3.2. *Choreography*

Earlier on, we distinguished the orientation of ports, i.e. we had different input and output actions, $s?[x]$ and $s![x]$, respectively. These are important for the interactions with actual providers of services. Since matching of processes is only concerned with control flow patterns in the choreography view, we ignore this distinction here, i.e. the composite role $s \circ [x]$ (or $s \circ x$) abstracts both $s?[x]$ and $s![x]$. Interaction does not need to be modelled further in an ontological form.

4.4. *Composition support*

4.4.1. *Matching*

The central inference technique in description logics is **subsumption**. Subsumption of concepts $C_1 \sqsubseteq C_2$ is the subset-relationship $C_1^I \subseteq C_2^I$ of the corresponding interpretations, i.e. the object classes. Equally, we define subsumption for roles $R_1 \sqsubseteq R_2$ as $R_1^I \subseteq R_2^I$. Before coming back to subsumption, we define service process matching – in the expected way.

> A provider process $P[n_1, .., n_k]$ **matches** a client process $C[m_1, .., m_l]$, if $P[n_1, .., n_k]$ simulates $C[m_1, .., m_l]$.

Subsumption on roles is input/output-oriented, whereas the simulation needs to consider internal states of the composite role execution. For each request in a process, there needs to be a corresponding provided service. Although clearly not the same, matching is a sufficient condition for subsumption [14]:

> If the process expression $P[n_1, \ldots, n_k]$ simulates the process $C[m_1, \ldots, m_l]$, then C is subsumed by P, i.e. $C \sqsubseteq P$.

Due to the same semantics as the operational framework, we can use the transition graphs approach presented earlier on to reason about simulation.

Example 9. Matching of processes – see Example 3 – is now also supported between ontological process expressions.

We formulate composition based on matching process descriptions in form of an inference rule in the next subsection.

4.4.2. *Connection and interaction*

The operational semantics of interaction can be defined in form of process calculus-style contract and connector rules – we have introduced some examples, see Fig. 8. In terms of the ontology, services were so far described as transitional roles and we considered system states that describe service (and process) properties such as pre- and post-states to define transitional process behaviour.

While we do not fully formalise composition and interaction in the ontology framework, we show how this can be achieved through inference rules of our description logic. We reformulate the original **contract rule** (Fig. 8), here without annotations:

$$\frac{m_C![m_I]; p_{m_C} \xrightarrow{m_C![m_I]} p_{m_C} \qquad n_C?[n_I]; p_{n_C} \xrightarrow{n_C?[n_I]} p_{n_C}}{m_C![m_I]; p_{n_C} + M_1 | n_C?[n_I]; p_{n_C} + M_2 \ \xrightarrow{\tau}\ p_{m_C} \frown p_{n_C}} \langle\ sign(n_C) = sign(m_C))$$

in terms of the ontology language through a description logic inference rule:

$$\frac{\forall\ m_C \circ m_I\ .\ post_{m_C} \qquad \forall n_C \circ n_I\ .\ post_{n_C}}{\forall\ m_C \circ m_I | n_C \circ n_I\ .\ post_{m_C} \sqcap post_{n_C}} \langle\ sign(n_C) = sign(m_C))$$

This inference rule for parallel composition complements other constructor-specific axioms and rules that we can derive from dynamic logic and process calculi, such as the axiom $\forall R; S.C \equiv \forall R.\forall S.C$ for the sequence operator. These axioms and inference rules form an application-specific extension of description logic that allows us to infer more properties about service processes and their interactions. In terms of the knowledge space, they lift the conceptual model to a logical theory for service processes.

5. Application in Semantic Service Engineering

Semantic Web services [20,21] are widely propagated as means of improving the discovery and composition of Web services. Ontologies are used to capture service properties. We demonstrate now that our service process ontology can be utilised within this semantic service context in order to demonstrate the applicability of our framework.

The central issue in the semantic services context is matching of individual services for service-level interactions, rather than the service process matching we looked at so far. The description of services normally includes behavioural aspects (e.g. pre- and postconditions), but also non-functional descriptions such as the author or a textual description of the service. We focus on abstract (functional) behaviour here. OWL-S [22] and WSMO [23] are examples of ontological frameworks that support matching of semantically described services. Both focus on the semantical description of services including abstract descriptions, quality-of-service aspects, and functional abstractions such as pre- and postconditions. Our approach, however, complements OWL-S and WSMO. OWL-S and WSMO represent services as concepts in the ontology, not as transitions. Therefore, the bridge to dynamic logic cannot be exploited directly. Dynamic logic is a logical framework that subsumes pre- and postcondition specification [11]. This allows us to integrate these service and service process contracts easily into our framework. Similar to signatures, we can associate (descriptive) pre- and postcondition roles to pre- and poststates, respectively (see Fig. 13).

5.1. *Contractual Description and Matching*

We can extend the service specification by contractual information to capture service semantics. We use pre- and postconditions as abstractions [24], enabling the design-by-contract approach [25].

Example 10. A requirements specification of a service user for a `Login` operation is:

operation `Login[id:ID,passwd:Pass]`
  pre `syntaxOK(id)`
  post `valid(id)` $\vee$ `invalid(id)` $\vee$ `unknown()`

An example of a service provider specification for a `UserLogin` service is:

operation `UserLogin[id:ID,passwd:Pass]`
  pre `true`
  post `valid(id)` $\vee$ `invalid(id)`

Two services described by pre- and postconditions and represented by contract ports n_C and m_C match, if n_C refines m_C:

> Contract port n_C **matches** m_C if n_C refines m_C, i.e. if the precondition is weakened and the postcondition is strengthened. In this case, we write $type_c(n_C) \leq type_c(m_C)$.

This definition is derived from the consequence rule of dynamic logic, which expresses refinement of programs [11,26]. The *type* notion here extends the signature notion *sign* we used earlier on. We have used a simple contract idea here to illustrate the technique; in practice a more advanced variant might be used [26].

Example 11. The provided service `UserLogin` matches the requirements of `Login` in Example 11. Operation `UserLogin` has

- a weaker, less restricted precondition (`syntaxOK(id)` implies `true`) and
- a stronger postcondition (the disjunction `valid(id)` $\vee$ `invalid(id)` implies `valid(id)` $\vee$ `invalid(id)` $\vee$ `unknown()`).

This means that the provided service satisfies the requirements; it is even better than requested.

Preconditions constitute provision declarations rather than requirements for the client. Consequently, clients often do not specify them in their strongest form.

5.2. *Ontological support*

Matching of services has been defined in terms of implications on pre- and postconditions of service operations, and has been represented as a subtype relation between the contract ports. Again, we want to integrate reasoning about services contract matching with subsumption. The **matching inference rule** for transitional roles shall be defined as follows:

$$\frac{\forall preCond.pre_P \sqcap \forall P.\forall postState.post_P}{\forall preCond.pre_C \sqcap \forall P.\forall postState.post_C} \left\langle \begin{array}{l} pre_P \sqsubseteq pre_C \\ post_C \sqsubseteq post_P \end{array} \right.$$

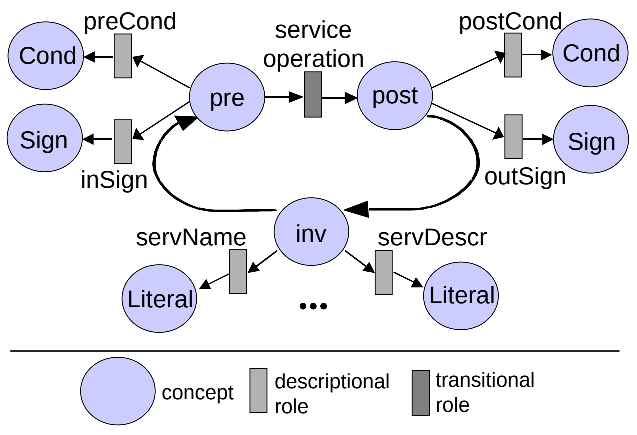

Fig. 13. Semantic Services Ontology.

Given a provided service operation P with corresponding pre- and postconditions, P can also be used in a context that is required by a potential client (in terms of pre- and postconditions for an abstract service operation C).

Matching implies subsumption, but is not the same. Matching of services is, however, a sufficient criterion for subsumption.

> If a service operation P matches (i.e. refines) a service operation R, then $P \sqsubseteq R$.

5.3. *Semantic service processes*

Our framework for semantic services matching has followed the philosophy of representing service-based software systems in form of a transition-oriented ontology; it has, however, not been fully integrated yet. In order to obtain a *semantic service process* framework based on *semantic services* and *service processes*, we need to add some support.

A notion of consistency of composite roles relates service processes to the underlying individual service specifications based on, for instance, pre- and postconditions. A concept description $\forall P[R_1, \ldots, R_n].C$ with composite transitional role P is **reachable** if the set of state transitions $\{(a,b) \in P^I | \exists b.b \in C^I\}$ is not empty. A composite role $P[R_1, \ldots, R_n]$ is **consistent**, if the last state is reachable through transitions. This abstract definition can be supported by a more constructive property. A composite transitional role P is consistent if the following (sufficient) conditions are satisfied:

- for each sequence $R; S$ in P: $\forall postCond.post_R \sqsubseteq \forall preCond.pre_S$
- for each iteration $!R$ in P : $\forall postCond.post_R \sqsubseteq \forall preCond.pre_R$

- for each choice $R + S$ in P: $\forall preCond.pre_R \sqcap \forall preCond.pre_S$ and $\forall postCond.post_R \sqcap \forall postCond.post_S$
- for each parallel composition $R|S$ in P: $\forall preCond.pre_R \sqcap \forall preCond.pre_S$ and $\forall postCond.post_R \sqcap \forall postCond.post_S$

We assume here that syntactical consistence is guaranteed, i.e. that for instance signatures match syntactically, if needed.

A **consistent service process** is a consistent composite role expression $P[R_1, \ldots, R_n]$ constructed from transitional role names $R_1, \ldots, R_n$ and the role connectors. In a combined semantic service process approach, consistency is a core requirement.

6. Application and Evaluation

The contribution of this paper is a two-layered, operational and ontological service process framework. In addition to the application to semantic service process engineering tasks, which we have described in the previous section, we now take a look at a broader range of specific applications and tool implementations and their evaluation.

6.1. *Applications*

Our contribution shall be evaluated using two application contexts to demonstrate the versatility of our approach: the first, service topology management, shall only utilise the operational process framework from Section 3; the second uses the ontological framework from Section 4 as a foundation for a conceptual service modelling and execution environment.

6.1.1. *Service topology management*

Service topology refers to the distributed architecture of service-based software systems and focuses on the functional and in particular non-functional characteristics as a result of the distributed locations in this architecture.

Service processes are at the core of service-oriented architectures. TOPMAN [30] is a model-driven development environment for service processes and the topologies they are embedded in. TOPMAN uses a model-driven approach in order to generate executable service interactions for service topologies based on UML-style process models. It allows these topologies to be configured based on abstract models rather than implementation descriptions.

TOPMAN is based on a process calculus to express service topology patterns as process abstractions. In order to clarify the topology of service processes, the notions of distribution and location need to be made explicit in topology models. In terms of our framework, the distinction between orchestration and choreography views on service systems need to be clarified. These topology aspects determine some of the crucial non-functional quality-of-service (QoS) properties such as efficiency, reliability and availability. TOPMAN uses the orchestration view, which is currently best supported by deployment platforms based on WS-BPEL. The operational model we presented here allows standard topology patterns to be described as service process abstractions. An example is the centralised topology pattern (also called Hub&Spoke)

$$\texttt{Centralised} \overset{\text{def}}{=} (\ \texttt{Hub?} \mid (\ \texttt{Spoke}_1\texttt{!} \mid \ \ldots \ \mid \texttt{Spoke}_n\texttt{!}\ )\ )*$$

which specifies the outer-level process of a centralised `Hub` receiving repeatedly invocation requests from a range of `Spokes`.

6.1.2. *Conceptual service modelling and execution*

While TOPMAN focuses on non-functional QoS properties, we have also investigated ontology-based conceptual modelling of functional properties [31], which we have combined with some service middleware functions.

The conceptual modelling environment is ontology-based [31]. It uses the service ontology from Section 4 as the formalisation of its UML-based graphical modelling notation. Moreover, the service ontology acts as a matchmaking engine using the matching rules defined earlier for both checking the correctness of abstract process composition and the discovery of semantically suitable services based on abstract service specifications – based on the semantic Web service application described in Section 5.

A UML-style process modelling notation, based on UML activity diagrams that can be annotated by semantic conditions, is the starting point. Using a recent ontology modelling standard, OMG's Ontology Definition Metamodel, we could integrate a UML-based modelling notation with an underlying ontology model thanks to metamodels and transformations provided by the ODM. Transformations between UML, OWL, and executable languages such as WS-BPEL support composition and code generation. This modelling environment is integrated with the GLORIS broker and execution engine [32] that allows distributed services to be located and invoked.

6.2. *Evaluation*

The core elements of our framework need to be evaluated according to their specific properties. Firstly, the operational process model needs to be looked at in terms of the soundness of the theoretical framework and its completeness with respect to the desired range of modelling support. The tractability of the theory is also an issue. Secondly, the ontological framework (based on the process foundations) needs to be looked at in terms of adequacy of the modelling notation and interoperability to enable integration with other service engineering techniques. Again, tractability is an issue. Additonally, the methodological integration with semantic services needs to be looked at in terms of its effectiveness as a methodology.

These applications of our foundational framework to develop the Web service development and deployment environments are supported by case studies:

- Banking: two aspects have been investigated: legacy systems integration and an online banking application. We have used a simplified online banking application throughout the paper to illustrate the notations and techniques we have introduced.
- E-learning: learning technology systems are knowledge-based applications that benefit from being exposed as semantically annotated service-oriented architectures.

We have developed tool prototypes to validate the central elements of our framework. Tool implementations complement and support the conceptual case studies in our evaluation of our framework as foundations for the core of a comprehensive service engineering solution.

The operational framework should be sound, tractable and complete in its provision of a description and modelling calculus:

- Soundness and tractability properties drectly base on those of the π-calculus. The notation and rules are a straightforward extension of the π-calculus that can easily be shown to be sound and tractable [12].
- The completeness of the modelling support is demonstrated by providing a standard range of basic actions and combinators – which is confirmed by our modelling case studies and process calculus literature [13].

The ontology framework aims to support semantics-based conceptual modelling, which requires adequacy, interoperability and tractability to be discussed as major desirable characteristics:

- The adequacy of the modelling notation is a major requirement. We have addressed this by carrying out different modelling case studies – two in the banking context and one in e-learning systems development [31]. All case studies have supported the notation as sufficient to address the various functional aspects of the respective system.
- Interoperability of models with other notations and tools is enabled through ODM, the Ontology Definition Metamodel [29], which defines a number of metamodels for UML, OWL, and other conceptual modelling notations, and transformations between them. This, for instance, allows to convert existing UML models into ontological representations that could be integrated into our approach, or to use UML as a visual front-end for ontologies as we have done it in our conceptual modelling application (Section 6.1.2).
- The tractability of the ontological framework is essentially determined by the decidability of the underlying description logic. We have positively asserted this property in [14] for the process ontology and its logical basis.

The methodological framework of the semantic service and service process integration needs to be demonstrated as being effective. Although we only presented an outline of such as methodology – our focus has been on foundations for notations and specific techniques – we have used and analysed service engineering and service-oriented architecture in different contexts. In particular the banking applications with a legacy integration project and a from-scratch development have provided valuable insights that have confirmed the benefits of service-based software development, but also the need to support these through semantic modelling and semantic annotations for discovery and execution of both services and processes.

7. Conclusions

The notion of processes is ubiquitous in computing – as a technical term for interacting systems or as an application-oriented one in business and workflow processes. Consequently, a wide range of formal models and modelling notations for process development has been devised.

Service-oriented architecture is a new architectural paradigm for software development. The current focus on usage (deployment, invocation, and reply) in these architectures has to be complemented by a more development-oriented one. Reuse of services in service assemblies is the ultimate development objective; process assembly is the principle of architectural composition of reusable services.

We have followed a layered approach to provide a basic semantic framework supporting a service engineering discipline. Firstly, we have presented an operational infrastructure model, focusing on service process interaction and composition, that facilitates development and deployment activities. Secondly, based on the infrastructure model, we have introduced an ontological framework to support development in the Web environment. An infrastructure model is needed as the underlying basis for an ontological layer that addresses aspects of this infrastructure. An ontological framework is needed to make process-oriented composition work as a development approach for a software developer for the Web platform with its emphasis on shared knowledge and joint activities. We have developed a basic framework based on ontologies, service-oriented architectures and processes, and underlying logics, applied to the Semantic Web and the Web Services Framework. In particular in the Web context, knowledge representation and knowledge sharing are becoming increasingly important for software development on the Web platform.

A central objective for both aspects – infrastructure and ontology – was to make the process characteristics as explicit as possible. We feel that the gap between metadata and annotation approaches (which are often captured ontologically) and operational process models and semantics has been to be narrowed in our framework, enabling a seamless integration of abstract description and process composition.

Acknowledgements. The authors are greatly indebted to Padraig de Buitlar, Rembert Douwen, and Gerard Gleeson, who all contributed tool components for the conceptual modelling and execution environment. The authors also wish to thank Olakunle Lemboye who carried out the analyses in the banking sector case study.

References

1. World Wide Web Consortium. *Web Services Framework.* http://www.w3.org/2002/ws, 2006. (visited 08/12/2006).
2. World Wide Web Consortium. *Web Services Choreography Description Language.* http://www.w3.org/TR/2004/WD-ws-cdl-10-20041217/, 2004. (visited 08/02/2006).
3. G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services – Concepts, Architectures and Applications.* Springer-Verlag, 2004.
4. F. Plasil and S. Visnovsky. Behavior Protocols for Software Components. *ACM Transactions on Software Engineering* 28(11):1056-1075. 2002.
5. L. Chen, N. Shadbolt, C.A. Goble, F. Tao, S.J. Cox, C. Puleston, and P.R. Smart. Towards a Knowledge-Based Approach to Semantic Service Composition. In D. Fensel, K.P. Sycara, and J. Mylopoulos, editors, *Proc. International Semantic Web Conference ISWC'2003.* Springer-Verlag, LNCS 2870, 2003.
6. J. Rao, P. Küngas, and M. Matskin. Logic-Based Web Services Composition: From Service Description to Process Model. In *International Conference on Web Services ICWS 2004*, pages 446–453. IEEE Press, 2004.
7. R. Zhang, I.B. Arpinar, and B. Aleman-Meza. Automatic Composition of Semantic Web Services. In *Proc. International Conference in Web Services ICWS'2003.* 2003.
8. N. Desai and M. Singh. Protocol-Based Business Process Modeling and Enactment. In *International Conference on Web Services ICWS 2004*, pages 124–133. IEEE Press, 2004.
9. C. Peltz. Web Service orchestration and choreography: a look at WSCI and BPEL4WS. *Web Services Journal*, 3(7), 2003.
10. J. Williams and J. Baty. Building a Loosely Coupled Infrastructure for Web Services. In *Proc. International Conference on Web Services ICWS'2003.* 2003.
11. D. Kozen and J. Tiuryn. Logics of programs. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B*, pages 789–840. Elsevier, 1990.
12. C. Pahl. A Formal Composition and Interaction Model for a Web Component Platform. In A. Brogi and E. Pimentel, editors, *Proc. ICALP Workshop on Formal Methods and Component Interaction.* Elsevier ENTCS Vol. 66 No. 4, 2002.
13. D. Sangiorgi and D. Walker. *The π-calculus – A Theory of Mobile Processes.* Cambridge University Press, 2001.
14. C. Pahl. An Ontology for Software Component Matching. *International Journal on Software Tools for Technology Transfer (STTT), Special Edition on Component-based Systems Engineering*, 7, 2006. (in press).
15. J.F. Sowa. *Knowledge Representation - Logical, Philosophical, and Computational Foundations.* Brooks/Cole, 2000.
16. M.C. Daconta, L.J. Obrst, and K.T. Klein. *The Semantic Web.* Wiley, 2003.
17. F. Baader, D. McGuiness, D. Nardi, and P.P. Schneider, editors. *The Description Logic Handbook.* Cambridge University Press, 2003.
18. K. Schild. A Correspondence Theory for Terminological Logics: Preliminary Report. In *Proc. 12th Int. Joint Conference on Artificial Intelligence.* 1991.
19. W3C Semantic Web Activity. Semantic Web Activity Statement, 2004. http://www.w3.org/2001/sw. (visited 06/12/2006).
20. T. Payne and O. Lassila. Semantic Web Services. *IEEE Intelligent Systems*, 19(4),

2004.

21. S. Agarwal, S. Handschuh, and S. Staab. Annotation, composition and invocation of semantic web services. *Journal of Web Semantics*, 2:31–48, 2004.
22. DAML-S Coalition. DAML-S: Web Services Description for the Semantic Web. In I. Horrocks and J. Hendler, editors, *Proc. First International Semantic Web Conference ISWC 2002*, LNCS 2342, pages 279–291. Springer-Verlag, 2002.
23. R. Lara, M. Stollberg, A. Polleres, C. Feier, C. Bussler, and D. Fensel. Web Service Modeling Ontology. *Applied Ontology*, 1(1):77–106, 2005.
24. J.B. Warmer and A.G. Kleppe. *The Object Constraint Language – Precise Modeling With UML*. Addison-Wesley, 1998.
25. Bertrand Meyer. Applying Design by Contract. *Computer*, pages 40–51, October 1992.
26. A. Moorman Zaremski and J.M. Wing. Specification Matching of Software Components. *ACM Trans. on Software Eng. and Meth.*, 6(4):333–369, 1997.
27. C. Pahl and M. Casey. Ontology Support for Web Service Processes. *In Proc. European Software Engineering Conference and Foundations of Software Engineering ESEC/FSE03.* ACM Press, 2003.
28. C. Pahl. A Conceptual Framework for Semantic Web Services Development and Deployment. *In L.-J. Zhang and M. Jeckle, editors: European Conference on Web Services ECOWS 2004.* Springer-Verlag. LNCS 3250. pp. 270-284. 2004.
29. Object Management Group. *Ontology Definition Metamodel - Request For Proposal (OMG Document: as/2003-03-40).* OMG, 2003.
30. R. Barrett, L. M. Patcas, J. Murphy, and C. Pahl. Model Driven Distribution Pattern Design for Dynamic Web Service Compositions. In *International Conference on Web Engineering ICWE06. Palo Alto, US.* ACM Press, 2006.
31. C. Pahl. Layered Ontological Modelling for Web Service-oriented Model-Driven Architecture. In *European Conference on Model-Driven Architecture ECMDA2005.* Springer LNCS Series, 2005.
32. G. Gleeson and C. Pahl. Service-based Grid Architectures to support the Virtualisation of Learning Technology Systems. In C. Pahl, editor, *Architecture Solutions for E-Learning Systems.* Idea Publishers, 2006. (in press)