WLFM 2005 Preliminary Version

A Semantical Framework for the Orchestration and Choreography of Web Services

Claus Pahl and Yaoling Zhu [1]

School of Computing
Dublin City University
Dublin 9, Ireland

Abstract

Web Services are software services that can be advertised by providers and invoked by customers using Web technologies. This concept is currently carried further to address the composition of individual services through orchestration and choreography to services processes that communicate and interact with each other. We propose an ontology framework for these Web service processes that provides techniques for their description, matching, and composition. A description logic-based knowledge representation and reasoning framework provides the foundations. We will base this ontological framework on an operational model of service process behaviour and composition.

Key words: Web services, choreography, orchestration, process model, ontologies.

1 Introduction

Service-oriented architectures (SOAs) provide an architectural paradigm for software development [1]. Systems can be organised in terms of services – units of software that provide functionality 'as is' to users. Functionality descriptions and other properties and quality attributes such as security or performance and usage-oriented information such as invocation protocols and locations are advertised by providers and can be looked up by potential users.

The Web Services Framework (WSF) is such an SOA [2]. The WSF provides an SOA infrastructure consisting of a description language (WSDL), an invocation protocol (SOAP), and a repository for descriptions (UDDI) based on standard Internet and Web technologies such as XML.

While the first generation of the WSF has focussed on the use of services 'as is', the next needs to address service composition to enable larger software

[1] Email: cpahl@computing.dcu.ie

systems to be assembled based on services as the basic unit [3,4]. Composition of services to processes is here the paradigm of composition. Two forms – orchestration and choreography – have recently been discussed in the WSF community as techniques for service composition and collaboration [5]. These two reflect the perspective of business processes modelled and executed (orchestration) and of systems as interacting processes (choreography).

So far, the WSF is focused more on invocation than development. UDDI supports potential users in locating suitable services; how these services are integrated into existing software systems and how these services can be composed to larger systems is, however, not sufficiently addressed. The state-of-the-art comprises languages for orchestration and choreography, such as WS-BPEL4 or WS-CDL [5]. The basis of these languages are workflow and message exchanges, and aspects of interaction processes and patterns. Principles of component-based software development CBSD [6] are not yet integrated.

We will therefore focus here on using the WSF platform as an infrastructure for service-based software systems development. The overall aim is to support (service-based) software development *on* and *for* the Web. Formal methods are proven to be successful for the development of safety-critical, dependable software systems. Formal models allow a higher degree of understanding of principles and mechanisms of the context, but also particular properties of the application. A formal model of orchestration and choreography and a description notation are therefore our central objectives.

An important requirement arises if in particular the Web-based development of service-based systems is to be realised. The Semantic Web paradigm needs to be embraced in order to support the SOA principle of distributed development involving different organisations. The semantic Web provides a shared knowledge representation framework and platform, based on ontologies at the core. Ontologies can capture properties of services; they can also support composition description and reasoning. Our objective here are:

- We clarify the notions of orchestration and choreography. To this end, we will provide a formal model for service process composition in Section 3.
- We provide an ontological framework for service process composition that supports the CBSD objective of reuse in Section 4. This framework will be based on the semantical definition of service process composition.

2 Development of Composite Service Processes

2.1 Web-based Service Development

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, but more and more the *composition of services* to *processes* is important.

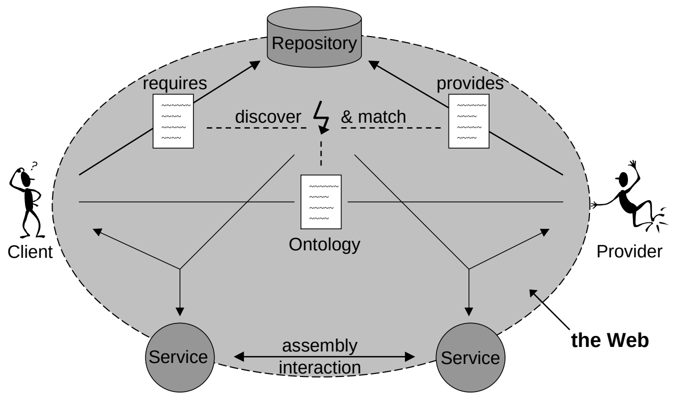

Fig. 1. The Web as a Service-oriented Development and Deployment Platform.

Reuse is a central software engineering principle. Existing services can be reused to form business or workflow processes. The *principle of architectural composition* here is *process assembly.*

The discovery and invocation infrastructure – a registry or marketplace where potential users can search for services and an invocation protocol – with the services and their clients form a *service-oriented architecture.* Languages for description and composition and protocols are central elements of this architecture. Fig. 1 illustrates this infrastructure for the WSF. Software development for service-oriented architectures is a two-step process. *Discovery* is based on abstract computation descriptions (and other software properties), formalised based on ontologies. *Assembly* is about composition of services to processes. Ontologies to represent knowledge about services is essential for the Web as a development platform. *Usage* complements a basic service *life cycle.* It is about communication and process interactions between services.

2.2 Service Process Composition – Orchestration and Choreography

The WSF provides a platform to invoke services on a 'usage as is'-basis. Real value, however, will be added if services can be connected [5]. Supporting and implementing business processes within the WSF through composed services is the requirement. Orchestration and choreography are two forms of service composition and collaboration that are currently discussed.

- *Orchestration* refers to a composed business process that may use both internal and external Web services to fulfill 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 and execution order.
- *Choreography* addresses the interactions that implement the collaboration between services. Multiple agents are considered where each agent describes its own part in the interaction.

Orchestration and choreography address different perspectives. Orchestration is focused on the internal behaviour of a business process. Choreography is focused on the external perspective, looking at process interaction. These

```
Service AccountProcess
  operation  import Login (no:int,user:string) :  bool
             import Balance (no:int) :  real
             import Lodgement (no:int,sum:real) :  void
             import Transfer (no:int,dest:int,sum:real) :  void
             import Logout (no:int) :  void
  process    $\overline{Login}$; !($\overline{Balance}$+$\overline{Lodgement}$+$\overline{Transfer}$);$\overline{Logout}$
Service BankAccount
  operation  export Balance (no:int) :  real
             export Lodgement (no:int,sum:real) :  void
             export Transfer (no:int,dest:int,sum:real) :  void
             import CheckAcc (dest:int) :  bool
  process    !(Balance+Logdement+(Transfer;$\overline{CheckAcc}$))
Service AccountRegistry
  operation  export CheckAcc (no:int) :  bool
  process    !CheckAcc
Service LoginServer
  operation  export Login (no:int,user:string) :  bool
             export Logout (no:int) :  void
  process    !(Login+Logout)
```

Fig. 2. Bank Account Processes and Services.

perspectives are the essential aspects of an SOA. We will capture these in a *process model* (orchestration) and an *interaction model* (choreography).

2.3 A Bank Account Example

An online banking example shall illustrate our service process framework. `Login;!(Balance+Lodgement+Transfer);Logout` is a process expression describing an interaction process of an online banking user starting with a login, then repeatedly executing balance enquiries, lodgements or money transfer, before loggin out. In Fig. 2, four services are described in a pseudo-code notation. Each of these services implements a process internally (orchestration). The interactions resulting from the service invocations (import, overlined, e.g. $\overline{\texttt{Balance}}$) and service provision (export, normal, e.g. `Login`) are the result of service choreography. For instance, `AccountProcess` is a client of `BankAccount` and `LoginServer`; `BankAccount` is a client of `AccountRegistry`.

3 Services and Processes – a Formal Operational Model

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

3.1 Orchestration and Choreography Description

3.1.1 Orchestration.

We can derive the following core requirements for an orchestration notation from languages such as WS-BPEL [5]:

- *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 to actions are needed.

The focus of orchestration is illustrated in Fig. 3. The business process itself and the Web services that implement the process are separated. This keeps the process logic apart from its implementation. The process is executed by an orchestration engine which invokes the respective services.

We capture the foundations of orchestration in form of a *process model* focussing on service composition. A process description is about control flow and the determination of the execution order. We will start with abstract actions to concentrate on control flow first – data aspects and also interactions will be added later. A process description can serve different purposes:

- to define a business process in terms of actions and control flow,
- 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).

Our process model is based on the principles and notation of the π-calculus.

Service process expressions, or **processes** are inductivley formed based on a basic process names, named process expressions, and the combinators sequence ; , parallel composition | , non-deterministic choice + , and iteration ! . A named process expression $P(s_1, \ldots, s_k)$ is defined by a service process expression on based services $s_1, \ldots, s_k$ and the combinators. The process definition is recursive. Based on basic processes (which are Web services), composite services 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.

Example 3.1 `Login; !(Balance+Lodgement+Transfer); Logout` is a business process for an online bank account.

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

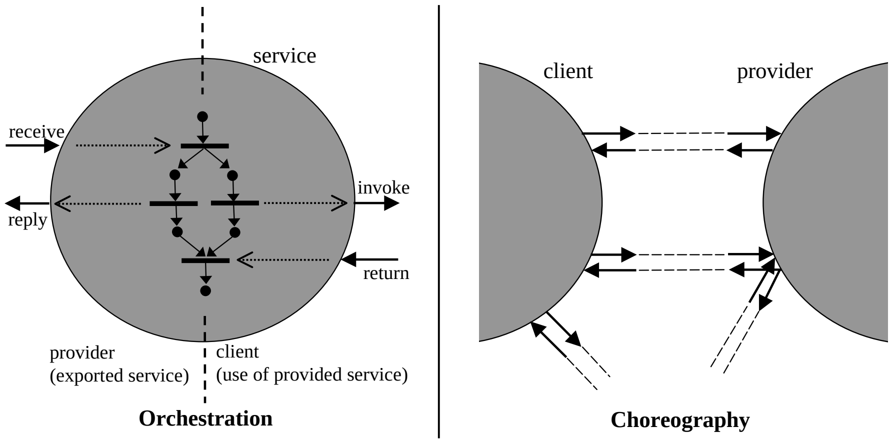

Fig. 3. Principles of Orchestraction and Choreography.

We now add data by refining the notion of actions. For service s and data item x, $s(x)$ is the **receive action**, $\overline{s}\langle x\rangle$ is the **reply action**, and *let* $y = \overline{s}\langle x\rangle$ *in* P is the **invoke action**. Receive and reply actions are needed to faciliate service provision. Invoke 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 note the main requirements for a choreography description notation [5]:

- *basic activities*: request/response action for local activities, invoke to call external services,
- *structured activities*: loop, sequence, choice, and concurrency,
- *infrastructure*: channels/connections between ports that represent services.

The focus of choreography is observable interaction behaviour, not execution, see Fig. 3. The orchestration model is a *process model* with its focus on control flow and execution order. The choreography model is an *interaction model* about process interaction, i.e. synchronisation and data exchange. Essential in modelling process interaction is to add data flow between processes.

Web services are connected through a network. The network endpoints that represent services are called *ports* – service names will 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. Assume a service port s and a data item x. Then, $s(x)$ is the **receive action** and $\overline{s}\langle x\rangle$ is the **send action**. Note, that in contrast to orchestration, we have abstracted here from the difference between provider actions (receive/reply) and client actions (invoke). The expression $\overline{\texttt{Balance}}\langle\texttt{acc}\rangle;\texttt{Balance}(\texttt{bal})$ asks service `Balance` for the current balance of account `acc` and then receives the balance `bal`.

An **interaction** is the activation of a remote service. Two forms shall be provided. Assume a process expression P.

- request-response: for each service s in P a write-read sequence $\overline{s}\langle x\rangle; s(y)$ where y is the returned result from an external service.
- execute-reply: for each service s in P a read-write sequence $s(x); \overline{s}\langle f(x)\rangle$ where f is some internal service functionality.

These interactions are the basic building blocks of the process life cycle. Input services 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 and managing the connections is part of the interaction model and its matching and connection support.

So far, the concurrent composition of processes $A|B$ 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 details in below. A *shared channel* can be created that forms a *connection* between two agents. Usually, the port names act as channel names (e.g. the π-calculus requires matching port names to establish a connection; we will loosen this constraint later on). Choreography is often about fixed connections. Process calculi, however, also cater for connections that are created dynamically. Using the π-calculus' scope extrusion, dynamic architectures can be modelled.

3.2 Composition Support

Descriptions are needed to publish services in repositories or to capture requirements for these services. We will provide a simple development and deployment model for services in form of a life cycle model, before addressing techniques needed for individual activities in that lifecycle.

3.2.1 Life Cycle and Activities.

Description and matching are design activities. Essential is, however, the support of the full process life cycle. Binding individual service names to existing services, i.e. composing a process instance and executing this instance are as important as description and matching. The foundations of these aspects are given in form of a choreography or *interaction model* that describes bindings, connections, and interactions between services.

Each service s is a family of ports s_C, s_I, s_R that address the needs of the different life cycle stages. 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

$$\text{REQ}\ \overline{s_C}\langle s_I\rangle;\ !(\ \text{INV}\ \overline{s_I}\langle a, s_R\rangle;\ \text{RES}\ s_R(y)\ )$$

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

$$\textsc{Pro}\ s_C(s_I);\ !(\ \textsc{Exe}\ s_I(a, s_R);\ \textsc{Rep}\ \overline{s_R}\langle f(a)\rangle\ )$$

with annotations for providing, executing and replying.

In the requestor view, $\textsc{Req}\ \overline{s_C}\langle s_I\rangle$ is an annotated output action of service s. A process 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 requestor port type and a provider port type is successful, then the requestor and the provider process can be composed, i.e. a requestor can interact with the provided service repeatedly. The requestor would invoke $\textsc{Inv}$ the service at port s_I and receive a result $\textsc{Res}$ at port s_R.

3.2.2 Matching.

Matching is central in composition. An existing service that is reused and integrated, for instance into a business process, must match the requirements in order to allow the business process to fulfill its task.

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

These 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.

The specification of processes describes the ordering of observable activities. We use a simulation notion to define process matching. The *requested process* is the import process pattern that the client expects the provider to support. A provider process P **matches** (or **simulates**) a requested process R if there exists a binary relation $\mathcal{S}$ over the set of processes such that if whenever $R\mathcal{S}P$ and $R \xrightarrow{m} R'$ then there exists P' such that $P \xrightarrow{n} P'$ and $R'\mathcal{S}P'$. This definition originates from the simulation definition of the π-calculus [7]. The provider needs to be able to simulate a request, i.e. needs to meet the request pattern of the client. Dynamic binding of concrete services to the process names is possible. This definition is about *potential* interaction.

Example 3.2 A provided service process `!(Balance+Lodgement+Transfer)` matches the expected support of process `!(Bal+Ldg)`. If the pairs `Balance`/`Bal` and `Lodgement`/`Ldg` match (e.g. equal signatures), then the provider matches (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 elements.

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 semantics.

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

For a parallel composition $\overline{m_C}\langle m_I\rangle.C | n_C(n_I).P$ of a client business process element and a provider service, both processes commit themselves to a communication along a (virtual) channel between ports m_C and n_C. A **contract rule** formalises the process of matching and commitment [2]:

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

Arrows denote state transitions of processes, either through observable actions $\overline{x}\langle y\rangle$ and $x(y)$ or through internal, non-observable *interactions* τ. We define a **composition** $C \frown P$ as $\nu c(\{c/m_I\}C | \{c/n_I\}P)$ [3], i.e. a parallel composition where a private channel c, the connector, replaces the port names.

Example 3.3 The user requires a service (annotation Req) through port $\texttt{Balance}_C$ $\texttt{Req} \overset{\text{def}}{=} \textsc{Req}\ \overline{\texttt{Balance}_C}\langle\texttt{Balance}_I\rangle; \texttt{Req}'$ and the server provides a service (annotation Pro) through port $\texttt{Bal}_C$ $\texttt{Pro} \overset{\text{def}}{=} \textsc{Pro}\ \texttt{Bal}_C(\texttt{Bal}_I); \texttt{Pro}'$.

A connector is created if a client requesting m_I invokes a service n_I at the server side, described by the **connector rule**:

$$\frac{\textsc{Inv}\ \overline{m_I}\langle a, m_R\rangle; C \xrightarrow{\overline{m_I}\langle a, m_R\rangle} C \quad \textsc{Exe}\ n_I(x, n_R); P \xrightarrow{n_I(x, n_R)} P}{\textsc{Inv}\ \overline{m_I}\langle a, m_R\rangle; C + M_1 | \textsc{Exe}\ n_I(x, n_R); P + M_2 \xrightarrow{\tau} C \frown \{a/x\}P} \langle\ \begin{array}{c} sign(n_I) = \\ sign(m_I) \end{array}$$

Parameter data a and a reply channel m_R are sent to the provider. Parameter a replaces x in P.

Example 3.4 The composition of `Pro'` and `Req'` creates a connector that allows the client with $\texttt{Req}' \overset{\text{def}}{=} \textsc{Inv}\ \overline{\texttt{Balance}_I}\langle\texttt{acc}\rangle; \texttt{Req}''$ to use a service, e.g. `Bal`, provided by the server $\texttt{Pro}' \overset{\text{def}}{=} \textsc{Exe}\ \texttt{Bal}_I(\texttt{no}); \texttt{Pro}''$. The requestor invokes (Inv) a service through interaction port $\texttt{Balance}_I$, which will trigger the execution (Exe) of $\texttt{Bal}_I$ with parameter `acc` by the server.

[2] This rule differs from the π-calculus reaction rule which requires channel names to be the same [7]. We only require equality of signatures. Type systems for the π-calculus usually constrain data that is sent; we constrain interaction between agents.

[3] The substitution $\{b/a\}$P means that b replaces a in P.

4 Services and Processes – an Ontological Framework

Supporting service development is ideally supported through ontology technology for shared representation of knowledge – here service descriptions. We illustrate how description and composition of services processes can be represented in a description logic that underlies a Web ontology language.

4.1 Ontologies for Web Services and Processes

The formal model (see Section 3) goes beyond what we need for the ontological framework to support the development of service-based software systems. Ontologies are needed to support composition through matching of patterns and processes, i.e. port orientation and other interaction and choreography aspects are not relevant since they address the deployment infrastructure. The ontological framework therefore abstracts the underlying formal operational model, which defines the development and deployment infrastructure. We will develop the ontological framework in terms of a description logic [8].

Description logic as the underlying logic of the Semantic Web is particularly interesting for the software engineering context due to a correspondence between description logic and dynamic logic (a modal logic of programs) [9]. This correspondence is based on a similarity between quantified constructors (expressing quantified relations between concepts) and modal constructors (expressing safety and liveness properties of programs).

4.2 A Basic Process Ontology

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

- **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 $\forall R.C$ – called **value restriction** – is interpreted based on either

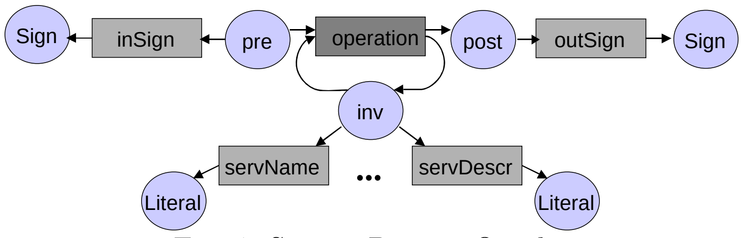

Fig. 4. Service Process Ontology

an accessibility relation R to a new state C for transitional roles, or on a property R satisfying a constraint C for descriptional roles. The dual $\exists R.C$ is called **existential quantification**.

In Fig. 4, the service process ontology is shown. A state is an abstract concept that is described in terms of elements of auxilary domains through descriptional roles such as invariant and mutable state properties (formal conditions, textual descriptions, etc.). The two essential state concepts 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`.

Throughout this paper, we use a description logic notation, e.g. for a given concept `pre`, we could constrain input signatures, $\forall$`inSign.(int,int)`. This notation is equivalent to a triple expression `(pre,insign,(int,int))`, which would be used in RDF (on which OWL is based).

4.3 Orchestration and Choreography

We introduced the representation of basic services in a description logic-based ontology. An ontology that captures service processes and their composition, however, requires an extension of classical description logics [8]. So far, roles – that represent service operations – are atomic. We define the combinators ';' , '!' , '|' and '+' as **role constructors** for sequential composition, transitive closure (iteration), intersection (parallel composition without interaction), and union (non-deterministic choice) of service processes, respectively. We also use $\circ$ for sequential composition to emphasise the functional character of roles. Role constructors allow us to integrate process description and composition into an ontology framework. The description logic expression $\forall$ `!(Balance+Lodgement+Transfer).post` describes a process.

Axioms and inference rules allow us to capture activity-related properties in the logic, e.g. in order to reason about matching. For example, $\forall R.\forall S.C \Leftrightarrow \forall R\,;S.C$ is an axiom that describes the conversion between logical operators and role expression combinators.

We need to integrate data and process parameters into the logic. We introduce data in form of *names*. Names stand for individual data elements.

- We denote a **name** n by a role n_N, interpreted by an identity relation $\{(n^I, n^I)\}$ for the interpretation n^I of n.
- An operation R is a **parameterised role** $R^I \subseteq \mathcal{D} \times \mathcal{S} \times \mathcal{S}$ for domain $\mathcal{D}$

of a name and states $\mathcal{S}$.

- A parameterised role R applied to a name n_N, represented here as an identity relation, i.e. $R \circ n_N$, forms a **transitional role**, i.e. $R \circ n_N \subseteq S \times S$[4].

Example 4.1 Given a transitional role `Login` and a descriptional role `outSign`, the expression $\forall$ `Login`$\circ$`(id`$_N$`,pwd`$_N$`)`.$\forall$`outSign.bool` means that by executing `Login`$\circ$`(id`$_N$`,pwd`$_N$`)` a state is reached that is described by a `boolean` result value. The term `Login`$\circ$`(id`$_N$`,pwd`$_N$`)` is a composite role expression in which the identifiers `id`$_N$ and `pwd`$_N$ are constant roles (names).

Earlier on, we distinguished the orientation of ports, i.e. we had different input and output actions, $s(x)$ and $\overline{s}\langle x\rangle$, respectively. These are important for the interactions with actual providers of services. Since matching of processes is here only concerned with control flow patterns, we ignore this distinction here, i.e. the composite role $s \circ x$ abstracts both $s(x)$ and $\overline{s}\langle x\rangle$. Interaction does not need to be modelled ontologically.

4.4 Composition Support

4.4.1 Matching.

Subsumption is the central inference technique in description logic. The **subsumption** $C_1 \sqsubseteq C_2$ of concepts is the subset-relationship of the corresponding object classes. Equally, we define subsumption for roles $R_1 \sqsubseteq R_2$. We define service process matching in the expected way. A process $P(n_1, .., n_k)$ **matches** a process $R(m_1, .., m_l)$, if $P(n_1, .., n_k)$ simulates $R(m_1, .., m_l)$. Subsumption on roles, however, 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 not the same, matching is a sufficient condition for subsumption. If the process expression $P(n_1, \ldots, n_k)$ simulates the process $R(m_1, \ldots, m_l)$, then $R \sqsubseteq P$. Matching can be ontologically supported by constructive axioms [10].

4.4.2 Connection and Interaction.

We have formulated the operational semantics of interaction in form of process calculus-style contract and connector rules. 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.

We formalise composition and interaction in the ontology framework through inference rules. In order to address interaction, we need to look at a special kind of a parallel composition transition. This transition is based on the *synchronisation* of concurrent services through data exchange. We can characterise properties of interactions between two services, here a reformulation

[4] We drop the $_N$-postfix when it is clear from the context that a name is referred to.

of the contract rule without annotations and matching constraints,

$$\frac{\overline{m_C}\langle m_I\rangle; p_{m_C} \xrightarrow{\overline{m_C}\langle m_I\rangle} p_{m_C} \qquad n_C(n_I); p_{n_C} \xrightarrow{n_C(n_I)} p_{n_C}}{\overline{m_C}\langle m_I\rangle; p_{n_C} + M_1 | n_C(n_I); p_{n_C} + M_2 \ \xrightarrow{\tau}\ p_{m_C} \frown p_{n_C}}$$

in terms of the ontology language by an inference rule:

$$\frac{\forall\ m_C \circ m_I\ .\ post_{m_C} \qquad \forall n_C \circ n_I\ .\ post_{n_C}}{\forall\ m_C \circ m_I | n_C \circ n_I\ .\ post_{m_C} \sqcap post_{n_C}}$$

This 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 $\forall p; q.C \equiv \forall p.\forall q.C$ for the sequence. These axioms and inference rules form an application-specific extension of description logic that allow us to infer more properties about service processes and their interactions.

5 Related Work

Composition of services is an active area of research [3,4,11]. In particular the need to address semantics in the context of composition has been recognised. In [3], an ontological framework for service composition is presented based on OWL-S (a rich services ontology, formerly known as DAML-S) as the underlying service ontology [13]. Their application area is the Grid services context and knowledge-based advice systems. OWL-S is different from our ontological framework in its process model. OWL-S represents services as concepts in the ontology, not as transitions. Therefore, the bridge to dynamic logics cannot be exploited in the way we proposed. Another OWL-S based approach is taken in [4]. Here, the logical side is strengthened. OWL-S descriptions are converted in linear logic and an architecture based on a logic-based planner and a semantic reasoner are proposed. The ultimate aim, as in our description-logic based approach, is the exploitation of logic reasoning for service composition.

Our approach differs from the discussed OWL-S-based approaches and other service ontologies such as WSMO [12] in that the ontological model captures services and processes in a more intrinisc way. OWL-S and WSMO address a wider range of properties, which suggests an integration of these approaches with our composition framework. We have aimed at reflecting the current discussion on orchestration and choreography in our technical and ontological models here.

Semantic Web services are a subject that our approach needs to be related to. OWL-S [13] and WSMO [12] 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.

We can use pre- and postconditions as abstractions for ports, enabling the design-by-contract approach [14]. Dynamic logic is a suitable logical framework that subsumes pre- and postcondition specification [15]. The connection between description logic and dynamic logic allows us to integrate these contracts easily into our framework. Similar to signatures, we can associate (descriptive) pre- and postcondition roles to pre- and poststates, respectively.

Two service operations described by pre- and postconditions and represented by contract ports n_C and m_C **match**, if the requestor's precondition is weakened and the postcondition strengthened [15,16]. Again, we would need to integrate reasoning about services matching with subsumption. Subsumption is interpreted as a subset relationship on sets of states that satisfy pre- or post-state descriptions. We present a matching inference rule for transitional roles. We define the **matching rule**

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

for transitional roles P and R. This form of matching implies subsumption, but is not the same: if service P matches R, then $P \sqsubseteq R$.

6 Conclusions

Organising software systems as service-oriented architectures is a new architectural paradigm. The Web Services Framework is currently the most important platform supporting this paradigm. In order to make the paradigm more successful as a software development approach, the focus on deployment, invocation, and reply has to shift towards more re-use and composition.

Service reuse is one of our objectives; process assembly of reusable services is the principle of architectural composition in this context. What is needed is a component-style composition framework for services – in particular an ontological framework to make it work as a development approach for distributed and shared platforms such as the Web. We have developed a formal framework based on ontologies (and underlying logics) and process-based service composition, applicable to the Semantic Web and Web Services platforms.

We have looked at core aspects of such a framework here – a complete and formal treatment was beyond the scope of this paper. One of the lessons we have learned is that a comprehensive formal framework for service-oriented architectures is needed to address predictable assembly, reuse, maintenance, and change and evolution management. We consider our contribution part of a methodology for service-based software development and deployment.

References

[1] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services – Concepts, Architectures and Applications.* Springer-Verlag, 2004.

[2] W3C – The World Wide Web Consortium. *Web Services Framework.* http://www.w3.org/2002/ws, 2004. (visited 08/04/2005).

[3] 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, *International Semantic Web Conference ISWC'03.* Springer LNCS 2870, 2003.

[4] 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.

[5] C. Peltz. Web Service orchestration and choreography: a look at WSCI and BPEL4WS. *Web Services Journal*, 3(7), 2003.

[6] C. Szyperski. *Component Software: Beyond Object-Oriented Programming – 2nd Ed.* Addison-Wesley, 2002.

[7] D. Sangiorgi and D. Walker. *The π-calculus – A Theory of Mobile Processes.* Cambridge University Press, 2001.

[8] F. Baader, D. McGuiness, D. Nardi, and P.P. Schneider, editors. *The Description Logic Handbook.* Cambridge University Press, 2003.

[9] K. Schild. A Correspondence Theory for Terminological Logics: Preliminary Report. In *Proc. 12th Int. Joint Conference on Artificial Intelligence.* 1991.

[10] C. Pahl. An Ontology for Software Component Matching. In M. Pezzè, editor, *Proc. Fundamental Approaches to Software Engineering FASE'2003*, pages 6–21. Springer-Verlag, LNCS 2621, 2003.

[11] 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.

[12] R. Lara, D. Roman, A. Polleres, and D. Fensel. A Conceptual Comparison of WSMO and OWL-S. In L.-J. Zhang and M. Jeckle, editors, *European Conference on Web Services ECOWS 2004*, pages 254–269. Springer-Verlag. LNCS 3250, 2004.

[13] 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.

[14] Bertrand Meyer. Applying Design by Contract. *Computer*, pages 40–51, October 1992.

[15] D. Kozen and J. Tiuryn. Logics of programs. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B.* Elsevier, 1990.

[16] A. Moorman Zaremski and J.M. Wing. Specification Matching of Software Components. *ACM Trans. on Software Eng. and Meth.*, 6(4):333–369, 1997.