An Ontology for Software Component Matching

Claus Pahl

Dublin City University, School of Computer Applications
Dublin 9, Ireland
Claus.Pahl@dcu.ie

Abstract. The Web is likely to be a central platform for software development in the future. We investigate how Semantic Web technologies, in particular ontologies, can be utilised to support software component development in a Web environment. We use description logics, which underlie Semantic Web ontology languages such as DAML+OIL, to develop an ontology for matching requested and provided components. A link between modal logic and description logics will prove invaluable for the provision of reasoning support for component and service behaviour.

1 Introduction

Component-based Software Engineering (CBSE) increases the reliability and maintainability of software through reuse [1,2]. Providing reusable software components and plug-and-play style software deployment is the central objective. CBSE originates from the area of object-oriented software development, but tries to overcome some of the problems associated with object-orientedness [1]. Components are software artefacts that can be individually developed and tested. Constructing loosely coupled software systems by composing components is a form of software development that is ideally suited for development in distributed environments such as the Web. Distributed component-based software development is based on component selection from repositories and integration.

Reasoning about component descriptions and component matching is a critical activity [3]. Ontologies are knowledge representation frameworks defining concepts and properties of a domain and providing the vocabulary and facilities to reason about these. Two ontologies are important for the component context:

- *Application domain ontologies* describe the domain of the software application under development.
- *Software development ontologies* describe the software development entities and processes.

The need to create a shared understanding for an application domain is long recognised. Client, user and developer of a software system need to agree on concepts for the domain and their properties. Domain modelling is a widely used requirements engineering technique. However, with the emergence of distributed software development and CBSE also the need to create a shared understanding of software entities and development processes arises. We will present here a

M. Pezzè (Ed.): FASE 2003, LNCS 2621, pp. 6–21, 2003.

software development ontology providing the crucial matching support for CBSE that is a substantial step ahead compared to the reasoning capabilities of current matching approaches such as DAML-S for Web Services [4,5].

Component matching techniques are crucial in Web-based component development. Providing component technology for the Web requires to adapt to Web standards. Since semantics are particularly important, ontology languages and theories of the Semantic Web initiative [6] need to be adopted. Formality in the Semantic Web framework facilitates machine understanding and automated reasoning. The ontology language DAML+OIL is equivalent to a very expressive description logic [7,8]. This fruitful connection provides well-defined semantics and reasoning systems. Description logics provide a range of class constructors to describe concepts. Decidability and complexity issues – important for the tractability of the technique – have been studied intensively.

Description logic is particularly interesting for the software engineering context due to a correspondence between description logics and modal logic [8,9]. The correspondence between description logics and dynamic logic (a modal logic of programs) is based on a similarity between quantified constructors (expressing quantified relations between concepts) and modal constructors (expressing safety and liveness properties of programs). We aim to enable the specification of transition systems in description logic. This enables us to reason about service and component behaviour. We present a novel approach to Web component matching by encoding transitional reasoning about safety and liveness properties – essentially from dynamic logic which is a modal program logic [10] – into a description logic and ontology framework.

We focus on the description of components and their services and their relation to the Semantic Web in Sect. 2. Reasoning about matching is the content of Sect. 3. We end with a discussion of related work and some conclusions.

2 Service and Component Description

2.1 The Component Model

Different component models are suggested in the literature [1,2,11,12]. Here is an outline of the key elements of our component model:

- *Explicit export and import interfaces.* In particular explicit and formal import interfaces make components more context independent. Only the properties of required services and components are specified.
- *Semantic description of services.* In addition to syntactical information such as service signatures, the abstract specification of service behaviour is a necessity for reusable software components.
- *Interaction patterns.* An interaction pattern describes the protocol of service activations that a user of a component has to follow in order to use the component in a meaningful way.

An example that illustrates our component model – see Fig. 1 – consists of a service requestor and a service provider component. The interface allows users to

Component DocInterface	*Component* DocStorageServer
import services	*import services*
create(id:ID)	...
retrieve(id:ID):Doc	*export services*
update(id:ID,upd:Doc)	crtDoc(id:ID)
preCond valid(upd)	rtrDoc(id:ID):Doc
postCond retrieve(id)=upd	updDoc(id:ID,upd:Doc)
export services	*preCond* wellFormed(upd)
openDoc(id:ID)	*postCond* rtrDoc(id)=upd∧wellFormed(upd)
saveDoc(id:ID, doc:Doc)	delDoc(id:ID)
import interaction pattern	*export interaction pattern*
create;!(retrieve+update)	crtDoc;!(rtrDoc+updDoc);delDoc

Fig. 1. Document processing example

open and save documents; it requires services from a suitable server component to create, retrieve, and update documents. The server provides a range of services. An empty document can be created using `crtDoc`. The request service `rtrDoc` retrieves a document, but does not change the state of the server component, whereas the update service `updDoc` updates a stored document without returning a value. Documents can also be deleted. A requirements specification of a service user for an `update` service is given. If documents are XML-documents, these can be well-formed (correct tag nesting) or valid (well-formed and conform to a document type definition DTD). We have specified an import interaction pattern for client `DocInterface` and for provider `DocStorageServer` an export pattern. The import pattern means that the `create` service is expected to be executed first, followed by a repeated invocation of either `retrieve` or `update`.

2.2 An Ontology for Component Description

The starting point in defining an ontology is to decide what the basic ontology elements – concepts and role – represent. Our key idea is that the ontology formalises a software system and its specification, see Fig. 2. Concepts represent component system properties. Importantly, systems are dynamic, i.e. the descriptions of properties are inherently based on an underlying notion of state and state change. Roles represent two different kinds of relations. *Transitional roles* represent accessibility relations, i.e. they represent processes resulting in state changes. *Descriptional roles* represent properties in a given state.

We develop a description logic to define the component matching ontology. A description logic consists of three types of entities. Individuals can be thought of as constants, concepts as unary predicates, and roles as binary predicates. Concepts are the central entities. They can represent anything from concrete objects of the real world to abstract ideas.

Definition 1. Concepts *are collections or classes of objects with the same properties. Concepts are interpreted by sets of objects.* **Individuals** *are named objects.* **Concept descriptions** *are formed according to the following rules: A is*

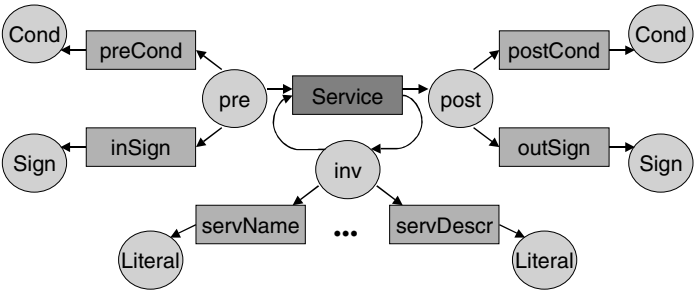

Fig. 2. Software development ontology

an atomic concept, and if C and D are concepts, then so are $\top$, $\bot$, $\neg C$, and $C \sqcap D$. Combinators such as $C \sqcup D$ or $C \rightarrow D$ are defined as usual. **Roles** *are relations between concepts.*

Roles allow us to associate properties to concepts. Two basic forms of role applications are important for our context. These will be made available in form of concept descriptions.

Definition 2. *Value restriction and existential quantification extend the set of concept descriptions*[1]*. A* **value restriction** $\forall R.C$ *restricts the value of role R to elements that satisfy concept C. An* **existential quantification** $\exists R.C$ *requires the existence of a role value. Quantified roles can be composed. Since $\forall R_2.C$ is a concept description, the expression $\forall R_1.\forall R_2.C$ is also a concept description.*

Example 1. An example for the value restriction is $\forall$`preCond.wellFormed`: all conditions are well-formed. An existential quantification is $\exists$`preCond.wellFormed`: there is at least one condition `preCond` that is well-formed.

The constructor $\forall R.C$ is interpreted as either an accessibility relation R to a new state C for transitional roles such as `update`, or as a property R satisfying a constraint C for descriptional roles such as `postCond`.

Example 2. For a transitional role `update` and a descriptional role `postCond`, the expression $\forall$`update.`$\forall$`postCond.equal(retrieve(id),doc)` means that by executing service `update` a state can be reached that is described by the post-condition `equal(retrieve(id),doc)` – an element of a condition domain.

2.3 Interpretation of Concepts and Roles

We interpret concepts and roles in Kripke transition systems [10]. Kripke transition systems are semantical structures used to interpret modal logics that are

[1] In description logic terminology, this language is called $\mathcal{ALC}$, which is an extension of the basic attributive language $\mathcal{AL}$.

also suitable to interpret description logics. Concepts are interpreted as states. Transitional roles are interpreted as accessibility relations.

Definition 3. *A* **Kripke transition system** $M = (\mathcal{S}, \mathcal{L}, \mathcal{T}, I)$ *consists of a set of states* $\mathcal{S}$*, a set of role labels* $\mathcal{L}$*, a transition relation* $\mathcal{T} \subseteq \mathcal{S} \times \mathcal{L} \times \mathcal{S}$*, and an interpretation* I*. We write* $R_{\mathcal{T}} \subseteq \mathcal{S} \times \mathcal{S}$ *for a transition relation for role* R*.*

The set $\mathcal{S}$ interprets the state domains *pre*, *post*, and *inv* – see Fig. 2. Later we extend the set $\mathcal{S}$ of states by several auxiliary domains such as *Cond*, *Sign*, or *Literal* that represent description domains for service and component properties.

Definition 4. *For a given Kripke transition system* M *with interpretation* I*, we define the model-based* **semantics of concept descriptions***:*

$$\begin{aligned}
\top &= \mathcal{S} \\
\bot &= \emptyset \\
(\neg A)^I &= \mathcal{S} \backslash A^I \\
(C \sqcap D)^I &= C^I \cap D^I \\
(\forall R.C)^I &= \{a \in S | \forall b.(a,b) \in R^I \rightarrow b \in C^I\} \\
(\exists R.C)^I &= \{a \in S | \exists b.(a,b) \in R^I \wedge b \in C^I\}
\end{aligned}$$

A notion of undefinedness or divergence exists in form of bottom $\bot$. Some predefined roles, e.g. the identity role *id* interpreted as $\{(x,x)|x \in \mathcal{S}\}$, shall be introduced. The predefined descriptional roles are defined as follows: $preCond^I \subseteq pre^I \times Cond^I$, $inSign^I \subseteq pre^I \times Sign^I$, $postCond^I \subseteq post^I \times Cond^I$, $outSign^I \subseteq post^I \times Sign^I$, $servName^I \subseteq inv^I \times Literal^I$, $servDescr^I \subseteq inv^I \times Literal^I$.

The semantics of description logics is usually given by interpretation in models. However, it can also be defined by translation into first-order logic [8]. Concepts C can be thought of as unary predicates $C(x)$. Roles R can be thought of as binary relations $R(x,y)$. Then, $\forall R.C$ corresponds to $\forall x.R(y,x) \rightarrow C(x)$.

2.4 Role Constructs and Component Behaviour

Expressive role constructs are essential for our application. *Transitional roles* $R_{\mathcal{T}}$ represent component services: $(R_{\mathcal{T}})^I \subseteq \mathcal{S} \times \mathcal{S}$. They are interpreted as accessibility relations on states. *Descriptional roles* $R_{\mathcal{D}}$ are used to describe properties of services dependant on the state: $(R_{\mathcal{D}})^I \subseteq \mathcal{S} \times \mathcal{D}$ for some auxiliary domain $\mathcal{D}$. These are interpreted as relations between states and property domains. In our case, the set of descriptive roles is fixed (*preCond*, *postCond*, *inSign*, *outSign*, etc.), whereas the transitive roles are application-specific services.

An ontology for component matching requires an extension of basic description logics by composite roles that can represent interaction patterns [8].

Definition 5. *The following* **role constructors** *shall be introduced:*

- $R\,;S$ **sequential composition** *with* $(R\,;S)^I = \{(a,c) \in S^I \times S^I | \exists b.(a,b) \in R^I \wedge (b,c) \in S^I\}$*; often we use* $\circ$ *instead of* $;$ *to emphasise functional composition*

- $!R$ **iteration** *with* $!R^I = \bigcup_{i \geq 1}(R^I)^i$*, i.e. the transitive closure of* R^I
- $R + S$ **non-deterministic choice** *with* $(R + S)^I = R^I \cup S^I$

Expressions constructed from role names and role constructors are **composite roles**. $P(R_1, \ldots, R_n)$ *is an abstraction refering to a composite role* P *based on the atomic roles* $R_1, \ldots, R_n$.

Example 3. The value restriction $\forall$ `create;!(retrieve+update) . postState` is based on the composite role `create;!(retrieve+update)`.

Definition 6. *A* **role chain** $R_1 \circ \ldots \circ R_n$ *is a sequential composition of functional roles (roles that are interpreted by functions).*

Axioms in our description logic allow us to reason about service behaviour. Questions concerning the consistency and role composition with respect to pre- and postconditions can be addressed.

Proposition 1. *Selected properties of quantified descriptions: (i)* $\forall R.\forall S.C \Leftrightarrow R;S.C$*, (ii)* $\forall R.C \sqcap D \Leftrightarrow \forall R.C \sqcap \forall R.D$*, (iii)* $\forall R \sqcup S.C \Leftrightarrow \forall R.C \sqcup \forall S.C$.

Proof. Follows from proofs from dynamic logic axioms such as $[\mathtt{p}][\mathtt{q}]\phi \Leftrightarrow [\mathtt{p;q}]\phi$ for (i) – see [10] Theorem 3. □

A special form of a role constructor is the existential predicate restriction. This will be needed in conjunction with concrete domains – see Sect. 2.6.

Definition 7. *The role expression* $\exists(u_1, \ldots, u_n).P$ *is an* **existential predicate restriction**, *if* P *is an* n*-ary predicate of a concrete domain – concepts can only be unary – and* $u_1, \ldots, u_n$ *are role chains. Analogously, we define the* **universal predicate restriction** $\forall(u_1, \ldots, u_n).P$.

$\exists(x, y).equal$ expresses that there are role fillers for the two roles x and y that are equal. The expression $\forall(x, y).equal$ requires all role fillers to be equal.

2.5 Names and Parameterisation

Individuals are introduced in form of assertions. For instance `Doc(D)` says that individual `D` is a document `Doc`. `length(D,100)` says that the length of `D` is 100.

Definition 8. Individual x *with* $C(x)$ *is interpreted by* $x^I \in \mathcal{S}$ *with* $x^I \in C^I$.

It is also possible to introduce individuals on the level of concepts and roles.

Definition 9. *The* **set constructor**, *written* $\{a_1, \ldots, a_n\}$ *introduces the individual names* $a_1, \ldots, a_n$*. The* **role filler** $R : a$ *is defined by* $(R : a)^I = \{b \in \mathcal{S} | (b, a^I) \in R^I\}$*, i.e. the set of objects that have* a *as a filler for* R.

This means that $R : a$ and $\exists R.\{a\}$ are equivalent.

The essential difference between classical description logic and our variant here is that we need names to occur in role and concept descriptions. A description logic expression $\forall$`create.valid` usually means that `valid` is a concept, or predicate, that can be applied to some individual object; it can be thought of as $\forall$`create(x).valid(x)` for an individual `x`. If roles are services, then `x` should not represent a concrete individual, but rather a name or a variable. For instance the document creation service `create` has a parameter `id`.

Our objective is to introduce names into the description language. The role filler construct provides the central idea for our definition of names.

Definition 10. *We denote a* **name** n *by a role* $n[Name]$*, defined by* $(n[Name])^I = \{(n^I, n^I)\}$*. A* **parameterised role** *is a transitional role* R *applied to a name* $n[Name]$*, i.e.* $R \circ n[Name]$*.*

In first-order dynamic logic, names are identifiers interpreted in a non-abstract state. These names would have associated values, i.e. a state is a mapping (binding of current values). However, since we are going to define names as roles this explicit state mapping is not necessary.

Proposition 2. *The name definition* $n[Name]$ *is derived from the role filler and the identity role definition:* $(n[Name])^I(n^I) = (id : n)^I$*.*

Proof. $(n[Name])^I(n^I) = \{(n^I, n^I)\}(n^I) = \{n^I\} = \{n^I | (n^I, n^I) \in id^I\} = \{b | (b, n^I) \in id^I\} = (id : n)^I$. □

The idea of presenting names as roles is borrowed from category theory[2].

We can now express a parameterised role $\forall$`create` $\circ$ `id`$[Name]$`.post` defined by $\{x | \forall y.(x, y) \in (\texttt{create} \circ \texttt{id}[Name])^I \rightarrow y \in \texttt{post}^I\}$ which is equal to $\{\texttt{id}^I | y \in \texttt{post}^I\}$, where y is a *postState* element that could be further described by roles such as $y = \forall$`postCond.post` $\sqcap \forall$`outSign.out`. The expression `create` $\circ$ `id`$[Name]$ is a role chain, assuming that `create` is a functional role: $(\texttt{create} \circ \texttt{id}[Name])^I = \{(a, c) | (a, b) \in \texttt{id}[Name]^I \wedge (b, c) \in \texttt{create}^I\} = \{(\texttt{id}^I, p) | (\texttt{id}^I, \texttt{id}^I) \in \texttt{id}[Name]^I \wedge (\texttt{id}^I, p) \in \texttt{create}^I\} = \{(\texttt{id}^I, p)\}$.

Example 4. With names and role composition the following *parameterised role chain* can now be expressed:

$\forall$ `update` $\circ$(`id`$[Name]$, `doc`$[Name]$); `postCond . equal(retrieve(id),doc)`

Note, that we often drop the $[Name]$ annotation if it is clear from the context that a name is under consideration.

[2] A point in category theory [13] resembles our name definition. A point in the category of finite sets is an arrow from a singleton set **1** to another object. This arrow can be seen as a mapping giving a name to a target value.

2.6 Concrete Domains and Property Types

Concrete domains and predefined predicates for these domains have been proposed to add more concrete elements to descriptions [8]. A classical example is to introduce a numerical domain with predicates such as $\leq$, $\geq$ or equality. These predicates can be used in the same way as concepts – which can also be thought of as unary predicates. An example is $\texttt{Doc} \sqcap \exists \texttt{length}.\geq 100$ where the last element is a predicate $\{n | n \geq 100\}$. `length` is a functional role, i.e. an attribute which maps to a concrete domain. Binary predicates such as equality can be used in conjunction with predicate restriction role constructors, e.g. $\exists(\texttt{x}, \texttt{y}).\texttt{equal}$.

Definition 11. Concrete domains *are interpreted by algebraic structures with a base set and predicates interpreted as n-ary relations on that base set.*

Concrete domains are important in our context since they allow us to represent application domain-specific knowledge. These domains will be refered to by type names. Concrete domains are needed for all application-oriented types used in a component specification.

Example 5. The update service deals with two types of entities: documents and identifiers. The *document domain* $\texttt{Doc} \equiv \exists \texttt{hasStatus}.\texttt{valid} \sqcup \texttt{wellFormed}$ and $\texttt{valid} \sqsubseteq \texttt{wellFormed}$ describes documents. Two predicates `valid` and `wellFormed` exist, which are in a subsumption or subclass relation. For the *identifier domain* `ID` only a binary predicate `equal` shall be assumed.

2.7 Contracts and Interaction Patterns

Axioms are introduced into description logics to reason about concept and role descriptions.

Definition 12. *Subconcept* $C1 \sqsubseteq C2$*, concept equality* $C1 \equiv C2$*, subrole* $R1 \sqsubseteq R2$*, role equality* $R1 \equiv R2$*, and individual equality* $\{x\} \equiv \{y\}$ *are* **axioms**. *The semantics of these axioms is defined based on set inclusion of interpretations for* $\sqsubseteq$ *and equality for* $\equiv$.

All forms of axioms are reducible to subsumption, i.e. subconcept or subrole [8]. Description logics often introduce an equivalence of concepts often as a definition in a macro style – the left-hand side is a new symbol, e.g. $\texttt{Status} \equiv \texttt{valid} \sqcup \texttt{wellFormed}$.

Contractual service descriptions form the basis of the matching of component services represented by atomic roles. The specification of `update` using axioms in description logic in Fig. 3 illustrates this. **Interaction patterns** can be specified using composite roles, e.g. $\forall \texttt{create} \circ \texttt{id}; !(\texttt{retrieve} \circ \texttt{id} + \texttt{update} \circ (\texttt{id}, \texttt{doc})).\texttt{post}$. It describes the interaction protocol that a component can engage in. There is one import interaction pattern and one export interaction pattern for each component.

The logic allows us to specify both safety and liveness properties of services.

$$
\begin{array}{lcl}
\texttt{pre} & \equiv & \forall \texttt{preCond.valid(doc)} \\
 & & \sqcap\ \forall \texttt{inSign.(id : ID, doc : Doc)} \\
 & & \sqcap\ \forall \texttt{update} \circ (\texttt{id}, \texttt{doc}).\texttt{post} \\
\texttt{post} & \equiv & \forall \texttt{postCond.equal(retrieve} \circ \texttt{id}, \texttt{doc}) \\
 & & \sqcap\ \forall \texttt{outSign.()} \\
\texttt{inv} & \equiv & \forall \texttt{servName.}\{\texttt{"update"}\} \\
 & & \sqcap\ \forall \texttt{servDescr.}\{\texttt{"updates document"}\} \\
 & & \sqcap\ \forall \texttt{update} \circ (\texttt{id}, \texttt{doc}).\texttt{inv}
\end{array}
$$

Fig. 3. Contractual description of service `update`

```
<daml:Class>
   <daml:Restriction>
       <daml:onProperty rdf:resource="#update(id,doc)"/>
       <daml:toClass>
           <daml:unionOf rdf:parseType="daml:collection">
               <daml:Restriction>
                  <daml:onProperty rdf:resource="#postCond"/>
                  <daml:hasClass rdf:resource="#equal(retrieve(id),doc)"/>
               </daml:Restriction>
               <daml:Restriction>
                  <daml:onProperty rdf:resource="#outSign"/>
                  <daml:hasClass rdf:resource="#()"/>
               </daml:Restriction>
           </daml:unionOf>
       </daml:toClass>
   </daml:Restriction>
</damL:Class>
```

Fig. 4. DAML+OIL specification

Example 6. We can express that eventually after executing *create*, a document will be deleted: $(\forall \texttt{preCond.true}) \sqcap (\forall \texttt{create.}\exists \texttt{delete.}\forall \texttt{postCond.true})$[3].

2.8 DAML+OIL and the Semantic Web

The Semantic Web initiative bases the formulation of ontologies on two Web technologies for content description: XML and RDF/RDF Schema. RDF Schema is an ontology language providing classes and properties, range and domain notions, and a sub/superclass relationship. Web ontologies can be defined in DAML+OIL – an ontology language whose primitives are based on

[3] This corresponds to a dynamic logic formula $\texttt{true} \rightarrow [\texttt{create(id)}]\langle \texttt{delete(id)} \rangle\, \texttt{true}$ combining safety ($[\ldots]\phi$) and liveness ($\langle \ldots \rangle \psi$) properties.

XML and RDF/RDF Schema, which provides a much richer set of description primitives. DAML+OIL can be defined in terms of description logics [14]. However, DAML+OIL uses a different terminology; corresponding notions are class/concept or property/role. We present the DAML+OIL specification of formula $\forall$`update` $\circ$ (`id`, `doc`).($\forall$`postCond.equal(retrieve(id),doc)` $\sqcap$ $\forall$`outSign.()`) in Fig. 4.

3 Inference and Matching

The two problems that we are concerned with are component description and component matching. Key constructs of description logics to support this are equivalence and subsumption. In this section, we look at component matching based on contracts and how it relates to subsumption reasoning.

3.1 Subsumption

Subsumption is defined by subset inclusions for concepts and roles.

Definition 13. *A* **subsumption** $C_1 \sqsubseteq C_2$ *between two concepts* C_1 *and* C_2 *is defined through set inclusion for the interpretations* $C_1^I \subseteq C_2^I$. *A* **subsumption** $R_1 \sqsubseteq R_2$ *between two roles* R_1 *and* R_2 *holds, if* $R_1^I \subseteq R_2^I$.

Subsumption is not implication. Structural subsumption (subclass) is weaker than logical subsumption (implication), see [8].

Proposition 3. *The following axioms hold for concepts* C_1 *and* C_2*: (i)* $C_1 \sqcap C_2 \sqsubseteq C_1$*, (ii)* $C_1 \wedge C_2 \rightarrow C_1$*, (iii)* $C_2 \rightarrow C_1$ *implies* $C_2 \sqsubseteq C_1$.

Proof. (i) $C_1 \sqcap C_2 \sqsubseteq C_1$ is true since $C_1^I \cap C_2^I \subseteq C_1^I$. *(ii)* $C_1 \wedge C_2 \rightarrow C_1$ is true since $(C_1 \wedge C_2)^I \subseteq C_1^I$. *(iii)* $C_2 \rightarrow C_1$ implies $C_2^I \subseteq C_1^I$ since structural subsumption is weaker than logical subsumption. □

We can use subsumption to reason about matching of two service descriptions (transitional roles).

3.2 Matching of Services

Subsumption is the central reasoning concept in description logics. We will integrate service reasoning and component matching with this concept.

A service is functionally specified through pre- and postconditions. Matching of services is defined in terms of implications on pre- and postconditions and signature matching based on the widely accepted design-by-contract approach[4]. The CONS inference rule, found in dynamic logic [10], describes the refinement of services. Based on the hypotheses $\phi \rightarrow \phi'$, $\phi \rightarrow [p]\ \psi$, and $\psi' \rightarrow \psi$ we can conclude $\phi' \rightarrow [p]\psi'$. A matching definition for services, i.e. transitional roles, shall be derived from the CONS rule.

[4] We ignore other descriptions such as invariants and possible improvements of our refinement notion through subsignatures here.

Definition 14. *A provided service P* **refines** *a requested service R, or service P* **matches** *R, if*

$$\frac{\forall inSign.in_R \sqcap \forall R.\forall outSign.out_R}{\forall inSign.in_P \sqcap \forall P.\forall outSign.out_P} \langle\ in_P \equiv in_R \wedge out_P \equiv out_R$$

(signatures are compatible: types of corresponding parameters are the same) and

$$\frac{\forall preCond.pre_R \sqcap \forall R.\forall postCond.post_R}{\forall preCond.pre_P \sqcap \forall P.\forall postCond.post_P} \langle\ pre_R \sqsubseteq pre_P \wedge post_P \sqsubseteq post_R$$

(requested service precondition is weakened and postcondition strengthened).

Matching of service descriptions is refinement. This is a contravariant inference rule that captures service matching based on formal behaviour specification.

Example 7. The service `updDoc` of the document server matches the requirements of `update` – a service that might be called in methods provided by the interface. Signatures are compatible. `updDoc` has a weaker, less restricted precondition (`valid(doc)` implies `wellFormed(doc)`) and a stronger postcondition (the conjunction `retrieve(id)=doc` $\wedge$ `wellFormed(doc)` implies `retrieve(id)=doc`). This means that the provided service satisfies the requirements.

Proposition 4. *The matching rule for services defined in Definition 14 is sound.*

Proof. (i) Assume that $\forall preCond.pre_R$ and $pre_R \sqsubseteq pre_P$. Then $preCond^I = \{(a,b)|b \in pre_R^I\}$ and $pre_R^I \subseteq pre_P^I$ implies $preCond^I = \{(a,b)|b \in pre_P^I\}$ for $\forall preCond.pre_P$. (ii) Assume that $\forall R; postCond.post_R$ and $post_P \sqsubseteq post_R$. The former implies that $(R; postCond)^I = \{(a,c)|(\exists b.(a,b) \in R^I \wedge (b,c) \in postCond^I) \wedge c \in post_R^I\}$ and $post_P^I \subseteq post_R^I$ implies that $(P; postCond)^I = \{(a,c)|(\exists b'.(a,b') \in P^I \wedge (b',c) \in postCond^I) \wedge c \in post_P^I\}$ for $\forall P; postCond.post_P$ for a role P that is syntactically compatible with R. □

Matching implies subsumption, but is not the same. Refinement (matching of services) is a sufficient criterion for subsumption.

Proposition 5. *If service P refines (or matches) R, then $P \sqsubseteq R$.*

Proof. If P refines R, i.e. $pre_R^I \subseteq pre_P^I$ and $post_P^I \subseteq post_R^I$, then for each $(a,b) \in P^I$ there is an $(a,b) \in R^I$. Therefore, $P^I \subseteq R^I$, and consequently $P \sqsubseteq R$. □

If the conditions are application domain-specific, e.g. predicates such as `valid(doc)`, then an underlying domain-specific theory provided by an application domain ontology can be integrated via concrete domains.

3.3 Matching of Interaction Patterns

Together with service matching based on contractual descriptions, interaction pattern matching is the basis of component matching.

A notion of consistency of composite roles relates to the underlying service specifications based on e.g. pre- and postconditions.

Definition 15. *A composite role* $P(R_1, \ldots, R_n)$ *is* **consistent**, *if the last state is reachable. A concept description* $\forall P(R_1, \ldots, R_n).C$ *with transitional role* P *is* **reachable** *if* $\{(a, b) \in P^I | \exists b.b \in C^I\}$ *is not empty.*

Proposition 6. *A composite role* P *is* **consistent** *if the following (sufficient) conditions are satisfied:*

(i) for each sequence $R;S$ *in* P : $\forall postCond.post_R \sqsubseteq \forall preCond.pre_S$
(ii) for each iteration $!R$ *in* P : $\forall postCond.post_R \sqsubseteq \forall preCond.pre_R$
(iii) for each choice $R + S$ *in* P : $\forall preCond.pre_R \sqcap \forall preCond.pre_S$ *and* $\forall postCond.post_R \sqcap \forall postCond.post_S$

Proof. By definition $(R;S)^I = \{(a, c) \in (R;S)^I | \exists b.(a, b) \in R^I \wedge (b, c) \in S^I\}$. Then $\forall postCond.post_R^I \subseteq \forall preCond.pre_S^I$ implies that each $b \in \forall postCond.post_R^I$ is also in $\forall preCond.pre_S^I$, i.e. $b \in \forall postCond.post_R^I \Rightarrow b \in \forall preCond.pre_S^I$. Similarly for $!R$ since $!R = R; \ldots; R$. For each $R + S$ both pre- and postconditions need to be enabled to guarantee successful execution. □

Definition 16. *A component* **interaction pattern** *is a consistent composite role* $P(R_1, \ldots, R_n)$ *constructed from transitional role names and the connectors* ; , | , *and* +[5]. *Interaction patterns are interpreted by* **pattern transition graphs** *for composite transitional roles, i.e. the graphs that represent all possible pattern executions.*

Both client and provider components participate in interaction processes based on the services described in their import and export interfaces. The client will show a certain import interaction pattern, i.e. a certain ordering of requests to execute provider services. The provider on the other hand will impose a constraint on the ordering of the execution of services that are provided.

The specification of interaction patterns describes the ordering of observable activities of the component process. Process calculi suggest simulations and bisimulations as constructs to address the equivalence of interaction patterns. We will use a notion of simulation between processes to define interaction pattern matching between requestor and provider.

Definition 17. *A provider interaction pattern* $P(S_1, \ldots, S_k)$ **simulates** *a requested interaction pattern* $R(T_1, \ldots, T_l)$, *or pattern* P **matches** R, *if there exists a homomorphism* μ *from the transition graph of* R *to the transition graph of* P, *i.e. if for each* $R_g \xrightarrow{T_i} R_h$ *there is a* $P_k \xrightarrow{S_j} P_l$ *such that* $R_g = \mu(P_k)$, $R_h = \mu(P_l)$, *and* S_j *refines* T_i.

[5] We often drop service parameters in patterns since only the ordering is relevant.

Matching of interaction patterns is simulation. The form of this definition originates from the simulation definition of the π-calculus, see e.g. [15]. Note, that simulation subsumes service matching. The provider needs to be able to simulate the request, i.e. needs to meet the expected interaction pattern of the requestor.

The definition implies that the association between S_i and T_j is not fixed, i.e. any S_i such that S_i refines T_j for a requested service T_j is suitable. For a given T_j, in principle several different provider services S_i can provide the actual service execution during the process execution.

Example 8. The provider requires `crtDoc;!(rtrDoc+updDoc);delDoc` and the requestor expects `create;!(retrieve+update)` as the ordering. Assuming that the service pairs `crtDoc`/`create`, `rtrDoc`/`retrieve`, and `updDoc`/`update` match based on their descriptions, we can see that the provider matches (i.e. simulates) the required server interaction pattern. The `delDoc` service is not requested.

As for service matching we expect interaction pattern matching not to be the same as subsumption. 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 pattern, there needs to be a corresponding provided service. However, matching is again a sufficient condition for subsumption.

Proposition 7. *If the component interaction pattern $P(S_1, \ldots, S_k)$ simulates the interaction pattern $R(T_1, \ldots, T_l)$, then $R \sqsubseteq P$.*

Proof. If $P(S_1, \ldots, S_k)$ simulates $R(T_1, \ldots, T_l)$, then for each $(a, b) \in R^I$ there is a pair $(a, b) \in P^I$. Therefore, $R^I \subseteq P^I$, and consequently $R \sqsubseteq P$ follow. □

Note, that the provider might support more transitions, i.e. subsumes the requestor, whereas for service matching, the requestor subsumes the provider (the provider needs to be more specific).

3.4 Complexity and Decidability

The tractability of reasoning about descriptions is a central issue for description logic. The richness of our description logic has some negative implications for the complexity of reasoning. However, some aspects help to reduce the complexity. We can restrict roles to functional roles. Another beneficial factor is that for composite roles negation is not required. We do not investigate this aspect in depth [8] – only one issue shall be addressed.

A crucial problem is the decidability of the specification if concrete domains are added. Admissible domains guarantee decidability.

Definition 18. *A domain D is called* **admissible** *if the set of predicate names is closed under negation, i.e. for any n-ary predicate P there is a predicate Q such that $Q^D = (S^D)^n \backslash P^D$, there is a name $\top_D$ for S^D, and the satisfiability problem is decidable; i.e. there exists an assignment of elements of S^D to variables such that the conjunction $\wedge_{i=1}^{k} P_i(x_1^{(i)}, \ldots, x_{n_i}^{(i)})$ of predicates P_i becomes true in D.*

Proposition 8. *We can show that our chosen concrete domains (documents and identifiers) – see Example 5 – are admissible.*

Proof. In [8], it is shown that the domain $\mathcal{N}$ with the set of nonnegative integers and the predicates $<, \leq, >, \geq$ is admissible. We can map documents and identifiers to nonnegative numbers and lexicographical ordering predicates to the binary predicates. Consequently, the domains are admissible. □

4 Related Work

The formula ∀`update ∘(id,doc).∀postCond.equal(retrieve(id),doc)` in description logic corresponds to [`update(id,doc)`][`postCond`] `retrieve(id)=doc` in dynamic logic. Schild [9] points out that some description logics are notational variants of multi-modal logics. This correspondence allows us to integrate modal axioms and inference rules about programs or processes into description logics. We have expanded Schild's results by addressing the problem of representing names in the notation and by defining a matching inference framework.

Some effort has already been made to exploit Semantic Web and ontology technology for the software domain [4,5,16]. All of these approaches have so far focused on the restricted component-aspects of Web services. [16] addresses the configuration of Web services; [5] presents solutions in the DAML-S context, which is the closest project to our work.

DAML-S [4] is a DAML+OIL ontology for describing properties and capabilities of Web services. DAML-S represents services as classes (concepts). Knowledge about a service is divided into two parts. A service profile is a class that describes what a service requires and what it provides, i.e. external properties. A service model is a class that describes workflows and possible execution paths of a service, i.e. properties that concern the implementation. DAML-S relies on DAML+OIL subsumption reasoning to match requested and provided services. DAML-S [4] provides to some extend for Web services what we aim at for Web components. However, the form of reasoning and ontology support that we have provided here is not possible in DAML-S, since services are modelled as concepts and not rules in the DAML-S ontology. Only considering services as roles makes modal reasoning about process behaviour possible.

5 Conclusions

Component development lends itself to development by distributed teams in a distributed environment. Reusable components from repositories can be bound into new software developments. The Web is an ideal infrastructure to support this form of development. We have explored Semantic Web technologies, in particular description logics that underlie Web ontology languages, for the context of component development. Ontologies can support application domain modelling, but we want to emphasise the importance of formalising central development activities such as component matching in form of ontologies.

Adding semantics to the Web is the central goal of the Semantic Web activity. Our overall objective has been to provide advanced reasoning power for a semantic Web component framework. We have presented description logic focussing on semantical information of components. The behaviour of components is essentially characterised by the component's interaction processes with its environment and by the properties of the individual services requested or provided in these processes. The reasoning capabilities that we have obtained and represented in form of a matching ontology go beyond current ontologies for service matching. Even though description logics have been developed to address knowledge representation problems in general, the connection to modal logics has allowed us to obtain a rich framework for representing and reasoning about components. Description logic is central for two reasons. Firstly, it is a framework focusing strongly on the tractability of reasoning, and, secondly, it is crucial for the integration of component technology into the Web environment.

Some questions have remained unanswered. Decidability and complexity results from description logic need to be looked at in more detail. We plan to adapt exiting proof techniques and to use description logic systems such as FaCT.

References

1. C. Szyperski. *Component Software: Beyond Object-Oriented Programming – 2nd Ed.* Addison-Wesley, 2002.
2. G.T. Leavens and M. Sitamaran. *Foundations of Component-Based Systems.* Cambridge University Press, 2000.
3. A. Moorman Zaremski and J.M. Wing. Specification Matching of Software Components. *ACM Trans. on Software Eng. and Meth.*, 6(4):333–369, 1997.
4. 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.
5. J. Peer. Bringing Together Semantic Web and Web Services. In I. Horrocks and J. Hendler, editors, *Proc. First International Semantic Web Conference ISWC 2002*, LNCS 2342, pages 279–291. Springer-Verlag, 2002.
6. W3C Semantic Web Activity. Semantic Web Activity Statement, 2002. `http://www.w3.org/sw`.
7. I. Horrocks, D. McGuiness, and C. Welty. Digital Libraries and Web-based Information Systems. In F. Baader, D. McGuiness, D. Nardi, and P.P. Schneider, editors, *The Description Logic Handbook.* Cambridge University Press, 2003.
8. F. Baader, D. McGuiness, D. Nardi, and P.P. Schneider, editors. *The Description Logic Handbook.* Cambridge University Press, 2003. (to appear).
9. K. Schild. A Correspondence Theory for Terminological Logics: Preliminary Report. In *Proc. 12th Int. Joint Conference on Artificial Intelligence.* 1991.
10. Dexter Kozen and Jerzy Tiuryn. Logics of programs. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B*, pages 789–840. Elsevier Science Publishers, 1990.
11. C. Pahl. Components, Contracts and Connectors for the Unified Modelling Language. In *Proc. Symposium Formal Methods Europe 2001, Berlin, Germany.* Springer-Verlag, LNCS-Series, 2001.

12. M. Casey. *Towards a Web Component Framework: an Investigation into the Suitability of Web Service Technologies for Web-based Components.* M.Sc. Dissertation. Dublin City University, 2002.
13. F.W. Lawvere and S. Schanuel. *Conceptual Mathematics.* Cambridge University Press, 1998.
14. DAML Initiative. *DAML+OIL Ontology Markup.* `http://www.daml.org,2001`.
15. R. Milner. *Communicating and Mobile Systems: the π-Calculus.* Cambridge University Press, 1999.
16. A. Felfernig, G. Friedrich, D. Jannach, and M. Zanker. Semantic Configuration Web Services in the CAWICOMS Project. In I. Horrocks and J. Hendler, editors, *Proc. First International Semantic Web Conference ISWC 2002*, LNCS 2342, pages 279–291. Springer-Verlag, 2002.