Int J Softw Tools Technol Transfer (2007) 9:169–178
DOI 10.1007/s10009-006-0015-9

SPECIAL SECTION ON FUNDAMENTAL APPROACHES TO SOFTWARE ENGINEERING

An ontology for software component matching

Claus Pahl

Published online: 5 July 2006
© Springer-Verlag 2006

Abstract Matching is a central activity in the discovery and assembly of reusable software components. We investigate how ontology technologies can be utilised to support software component development. We use description logics, which underlie Semantic Web ontology languages, such as OWL, 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 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. 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 World-Wide Web. Distributed component-based software development is based on component selection and matching from repositories and their integration.

Reasoning about component descriptions and component matching is a critical activity [3]. Ontologies, which are knowledge representation frameworks defining concepts and properties of a domain and providing the vocabulary and facilities to reason about these, can support this activity.

C. Pahl (✉)
Dublin City University, School of Computing
Dublin 9, Ireland
e-mail: Claus.Pahl@dcu.ie

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, the need to create a shared understanding of software entities and development processes also arises. We will present here a software development ontology that provides matching support for CBSE [4,5].

Component matching techniques are crucial in Web-based component development. As far as matching is concerned, Web services exhibit component character. To provide component technology for the Web requires adaptation to Web standards. Since semantics are particularly important, ontology languages and theories of the Semantic Web [6] can be adopted. Formality in the Semantic Web framework facilitates machine understanding and automated reasoning. The Web ontology language OWL is equivalent to a very expressive description logic [7]. 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 [7].

Description logic is particularly interesting for the software engineering context due to a correspondence between description logics and modal logics [7,8] – modal logics have been used extensively to address temporal and behavioural aspects of state-based software systems. The correspondence between description logics and dynamic logic (a modal logic of programs,

[9] 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 facilitate the specification of state-based transition systems in description logic. This enables us to reason about component behaviour. We present an approach to component matching by encoding transitional reasoning about safety and liveness properties – essentially from dynamic logic – into a description logic and ontology framework, which is Web standards-compliant and has the benefit of tractability.

We introduce our component composition framework in Sect. 2. We focus on the description of components in an ontological framework in Sect. 3. Reasoning about matching is the content of Sect. 4. We end with a discussion of related work and some conclusions.

2 Component-based development

A compositional approach is important for distributed software development. Description, matching, and assembly are central activities in the distributed context. Formal, ontology-based support is ideal for this context due to its sharing and agreement aims.

2.1 The component model

A component is a set of operations provided as a reusable, highly context-independent software artefact. A component model defines core properties of a component. Different component models are suggested in the literature [1,2]. We capture common key elements in our component model for a distributed context:

- *Explicit export and import interfaces*. In particular explicit and formal import interfaces make components more context independent. Only properties of required components and operations are specified.
- *Semantic description of operation behaviour*. In addition to syntactical information, the abstract specification of functional behaviour of operations is a necessity for reusable software components. In a design-by-contract style [11], abstract behaviour can be expressed through pre- and postconditions.
- *Component interaction protocols*. An interaction protocol describes the ordering of operation activations that a component user has to follow to use the component meaningfully and consistently; for instance an object creation might be required before any inspection or modification can be carried out.

Syntax, operation semantics, and interaction protocols form an extended *contract* notion.

2.2 An ontology-based development framework

Ontologies capture knowledge about a domain in terms of concepts and roles. Concepts are described in terms of their relationships to other concepts through roles. Knowledge is divided into two forms: intensional and extensional. Intensional knowledge is general and abstract, captured through concepts and roles. Extensional knowledge refers to application-specific individuals relating to the concepts and roles. Two aspects of ontologies can be distinguished. Firstly, the *terminological aspect* defines a description notation. Secondly, the *logical aspect* provides a reasoning framework that can, for instance, support component matching.

Two types of ontologies are important in the context of component development and deployment:

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

A developer selects required components from ontological descriptions found in repositories, Fig. 1. Descriptions of required and provided components need to be matched. In an open, wide-area context, an accepted ontology-based description format and matching techniques are prerequisites.

2.3 Case study

The context of our case study is a *document storage service* for XML-based documents – which could be thought of as an abstraction of a database for XML-documents.

A sample specification in a pseudocode representation that illustrates our component model, see Fig. 2. It consists of a service requestor/user and a service provider component. The service user requires (imports) operations from a suitable server component to create, retrieve, and update documents. The server provides (exports) a range of operations in form of a component. An empty document can be created using `crtDoc`. The operation `rtrDoc` retrieves a document, but does not change the state of the server component, whereas the update operation `updDoc` updates a stored document without returning a value. Documents can also be deleted. The `update` and `updDoc` operation s are semantically specified through pre- and postconditions. XML- documents can be well-formed (correct tag

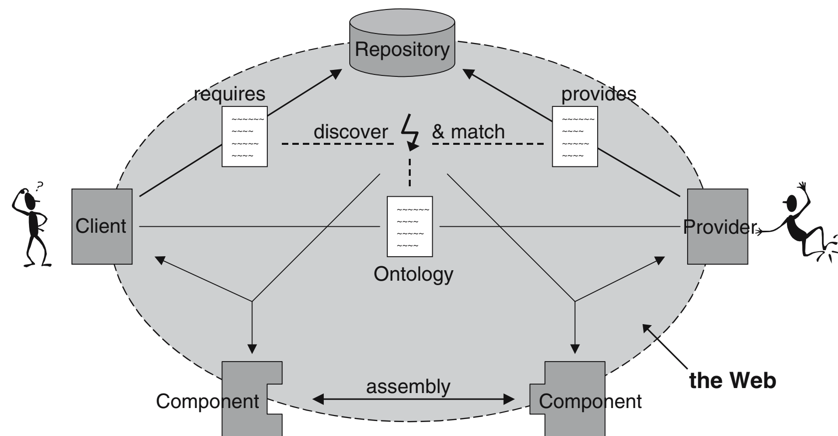

Fig. 1 Web-based Component Development Lifecycle based on Discovery/Matching and Assembly

nesting) or valid (well-formed and conform to an XML Schema definition). We have specified an import interaction protocol for client `DocStorageUser` and for provider `DocStorageServer` an export protocol. The import pattern means that `create` is expected to be executed first, followed by a repeated invocation of either `retrieve` or `update`.

3 An ontology for component description

A central objective of ontologies is the definition of a terminological framework. In this section, we define the syntax and semantics of a component description language in an ontological setting.

Our component description and matching ontology is non-standard, with features that go beyond classical knowledge representation. We will develop this ontology now step by step, demonstrating how the ontological features support component description.

3.1 Describing basic component properties

Ontologies formalise knowledge about a domain (intensional knowledge) and its instances (extensional knowledge). The starting point in defining an ontology is to decide what the basic ontology elements – concepts and roles – represent. Our key idea is that the ontology formalises a software system and its specification, see Fig. 3. Concepts represent component system properties. Importantly, component 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* address the state-transition aspect of software systems. They are interpreted as accessibility relations on states, i.e. they model behaviour as transitions resulting in state changes.
– *Descriptional roles* capture knowledge about components in form of description domains, i.e. they represent different properties of a software system. They cover syntax (signatures) and semantics (pre- and postconditions) of operations; they also capture state-dependent and invariant properties (informal descriptions, e.g. the component author).

We develop a description logic to define the component description and 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. Roles relate concepts with another.

– **Concepts** are classes of objects with the same properties. Concepts are interpreted by sets of objects.
– **Roles** are relations between concepts. Roles allow us to define a concept in terms of other concepts.
– **Individuals** are named objects.

Properties are specified as **concept descriptions**:

– **Basic concept descriptions** are formed according to the following rules: A is an atomic concept, and if C and D are concepts, then so are $\neg C$ (negation), $C \sqcap D$ (conjunction), $C \sqcup D$ (disjunction), and $C \rightarrow D$ (implication).
– Value restriction and existential quantification, based on roles, extend the set of basic concept descriptions:
  – 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, e.g. $\forall R_1.\forall R_2.C$ is a concept description since $\forall R_2.C$ is one.

Example 1 An example of a value restriction is the expression $\forall$`preCond.wellFormed`: preconditions

```
Component DocStorageUser                    Component DocStorageServer

  import operations                           import operations
    create(id:ID)                               ...           % not relevant here
    retrieve(id:ID):Doc                       export operations
    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 operations                               preCond    wellFormed(upd)
    ...           % not relevant here             postCond   rtrDoc(id)=upd∧wellFormed(upd)
  import interaction protocol                   delDoc(id:ID)
    create;!(retrieve+update)                 export interaction protocol
                                                crtDoc;!(rtrDoc+updDoc);delDoc
```

Fig. 2 Document Storage Service Example with Client (`DocStorageUser`) and Provider (`DocStorageServer`) Components

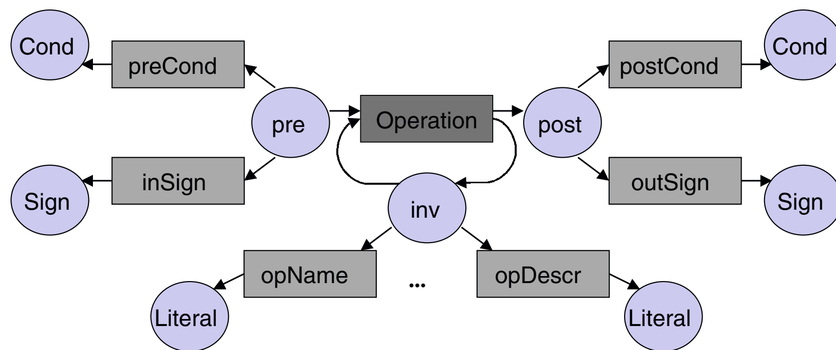

Fig. 3 Software Development Ontology based on Transitional Roles (Operation) and Descriptional Roles (preCond, inSign, etc.)

associated to a given concept (such as an operation) using role `preCond` are restricted to well-formed ones. The existential quantification ∃`preCond.wellFormed` requires 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 Given the transitional role `update` that represents a component operation and the descriptional role `postCond`, the expression

∀`update`.∀`postCond.equal(retrieve(id),doc)`

means that by executing operation `update` a poststate described by `equal(retrieve(id),doc)` as the postcondition can be reached [1]. □

We define our language through Tarski-style model semantics. We interpret concepts and roles in Kripke transition systems [9]. The concepts *pre*, *post*, and *inv* are interpreted as states, denoting prestates, poststates, and invariant state properties, respectively. Transitional roles are interpreted as accessibility relations between pre- and poststates, while descriptional roles are interpreted as associations between states and description domains.

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

[1] We ignore here the necessary parameterisation of `update` – which we will address in Sect. 3.4 and Example 7.

$R_{\mathcal{T}} \subseteq \mathcal{S} \times \mathcal{S}$ for a transition relation for role R. The set $\mathcal{S}$ interprets the state domains *pre*, *post*, and *inv* – see Fig. 3. We extend $\mathcal{S}$ by description domains *Cond* (conditions/formulas), *Sign* (signatures), and *Literal* for non-functional component properties.

For a given Kripke transition system M with interpretation I, we define the model-based **semantics of concept descriptions** as follows[2]:

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

An **individual** x defined by $C(x)$ is interpreted by $x^I \in \mathcal{S}$ with $x^I \in C^I$. A notion of undefinedness or divergence can be defined as bottom $\bot = \emptyset$. Some predefined roles, e.g. the identity role *id* interpreted as $\{(x,x)|x \in \mathcal{S}\}$, shall be assumed.

The **descriptional roles** are defined as relations between states and description domains:

$$
\begin{array}{ll}
preCond^I & \subseteq pre^I \times Cond^I \\
inSign^I & \subseteq pre^I \times Sign^I \\
postCond^I & \subseteq post^I \times Cond^I \\
outSign^I & \subseteq post^I \times Sign^I \\
opName^I & \subseteq inv^I \times Literal^I \\
opDescr^I & \subseteq inv^I \times Literal^I
\end{array}
$$

Note, that, while descriptional roles are predefined, transitional roles depend on the application.

3.2 Data types and concrete domains

We have introduced a number of predefined *description domains* capturing various forms of knowledge about a component. Formally, these are concepts representing formulas, signatures, etc. These capture only the syntactical correctness of the description, i.e. whether a string is actually a formula or signature.

In order to allow data to be modelled, we use *concrete domains* and *predefined predicates* [7] for these domains to add a notion of *data types* that can be linked to description domains such as formulas and signatures.

Example 3 We can 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 be thought of as unary predicates.

A case study example is $\texttt{Doc} \sqcap \exists \texttt{length}. \geq 100$ where the last element is a predicate $\{n|n \geq 100\}$ and `length` is a descriptional role, i.e. an attribute which maps to a concrete domain. □

[2] Combinators $\sqcap$ and $\rightarrow$ can be defined based on $\sqcup$ and $\neg$ as usual.

A special form of role constructors helps us in expressing n-ary predicates:

- 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 roles.
- Analogously, we define the **universal predicate restriction** $\forall(u_1,\ldots,u_n).P$.

Example 4 $\exists(\texttt{x},\texttt{y}).\texttt{equal}$ is a binary predicate restriction requiring role instances for the two roles x and y to be equal; for instance in- and outsignatures could be compared through $\exists(\texttt{inSign},\texttt{outSign}).\texttt{equal}$.

Concrete domains are interpreted by algebraic structures with a base set; predicates are interpreted as n-ary relations on that base set. Concrete domains are important here since they allow us to represent application domain-specific knowledge in a component specification. These domains will be referred to by type names.

Example 5 The `update` operation deals with two types of entities:

- The *document domain* $\texttt{Doc} \equiv \exists \texttt{hasStatus}.\texttt{valid} \sqcup \texttt{wellFormed}$ with $\texttt{valid} \sqsubseteq \texttt{wellFormed}$ defines documents, using `hasStatus` as a document attribute that associates a status. Two predicates `valid` and `wellFormed` exist, which are in a so-called subsumption, i.e. subclass relation.
- For the *identifier domain* `ID` only the binary predicate `equal` shall be assumed. □

We do not integrate and axiomatise a full first-order predicate logic here to support the data type domains. Instead, we assume that required properties are made available for the description logic through *assertions* [7]. Ontologies capture general intensional knowledge on a terminological level and extensional knowledge about concrete individuals. The assertions are part of the extensional, application-specific knowledge.

3.3 Functional behaviour and interaction protocols

Expressive role constructs are essential for our context. *Transitional roles* R_{T} represent component operations. They are interpreted as accessibility relations on states $(R_{\mathrm{T}})^I \subseteq \mathcal{S} \times \mathcal{S}$. *Descriptional roles* R_{D} are used to describe properties of operations. These are interpreted as relations between states and description domains $(R_{\mathrm{D}})^I \subseteq \mathcal{S} \times \mathcal{D}$ for some domain $\mathcal{D}$.

An ontology for component description requires an extension of basic description logics by composite roles

in order to represent interaction protocols [7]. The following **role constructors** for transitional roles shall be introduced to model interaction protocols:

- $R\,;S$ is **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
- $!R$ is **iteration** with $!R^I = \bigcup_{i \geq 1} (R^I)^i$, i.e. the transitive closure of R^I
- $R + S$ is **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** referring to a composite role P based on basic roles $R_1, \ldots, R_n$. A **role chain** $R_1 \circ \ldots \circ R_n$ is a sequential composition of functional roles[3].

Example 6 The value restriction

$\forall$ `create;!(retrieve+update).postState`

is based on the composite role

`create;!(retrieve+update)`

which is a required interaction protocol, see Fig. 2. □

3.4 Names and parameterisation

A notion of parameterisation for component operations is lacking so far in our ontological description language.

Named individuals might serve as parameter names. Individuals are introduced in form of assertions, e.g. `Doc(D)` says that individual `D` is a document `Doc` and `length(D,100)` that the length of `D` is 100. We can also introduce individuals on the level of concepts and roles:

- 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.

The difference between classical description logic and our variant is that we need names to occur explicitly in component descriptions. An intensional description logic expression $\forall$`create.valid` 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`. In the context of parameterisation, `x` should rather be an intensional name or variable, e.g. the document `create`–operation has a parameter called `id`. The role filler construct provides the central idea for our definition of names.

- We denote a **name** n of a domain $\mathcal{D}$ by a role $\underline{n}_N$ – i.e. not as an element of a concrete domain – where $\underline{n}_N$ is defined by $(\underline{n}_N)^I = \{(n^I, n^I)\}$ with $n^I \in \mathcal{D}^I$.
- 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 $\underline{n}_N$, represented here as an identity relation, i.e. $R \circ \underline{n}_N$, forms a **transitional role**, i.e. $R \circ \underline{n}_N \subseteq S \times S$.

The name definition $\underline{n}_N$ is derived from the role filler and the identity role definition: $(\underline{n}_N)^I(n^I) = (id : n)^I$.

In first-order dynamic logic [9], names are identifiers interpreted in a non-abstract state. These names have associated values, i.e. a state is a mapping (binding of current values). However, since we define names as roles, an explicit state mapping is not necessary.

Example 7 The parameterised role chain

$\forall$ `update` $\circ$ ($\underline{\texttt{id}}_N$, $\underline{\texttt{doc}}_N$);`postCond`.
`equal(retrieve(id),doc)`

specifies the component operation `update`.

3.5 Contractual operation and protocol specification

The original case study specification in pseudo-code (Fig. 2) needs to be reformulated in terms of the ontology language we have developed. **Axioms** are introduced into description logics to capture concept and role descriptions and to reason about these [7]:

- subconcept $C1 \sqsubseteq C2$, concept equality $C1 \equiv C2$,
- subrole $R1 \sqsubseteq R2$, role equality $R1 \equiv R2$, and
- individual equality $\{x\} \equiv \{y\}$.

The semantics of these axioms is defined based on set inclusion of interpretations for $\sqsubseteq$ and equality for $\equiv$.

We use axioms to formulate two different kinds of *component contract* specifications – operation behaviour and interaction protocols:

- **Functional behaviour** and **signatures** form the basis of a matching notion for component operations, which are represented by atomic roles.

  Example 8 The `update` specification based on description logic illustrates an operation definition in terms of our ontology, see Fig. 4 illustrates this.

[3] Functional roles are transitional roles that are interpreted by functions.

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

Fig. 4 Ontological specification of operation `update`

– **Interaction protocols** for components can be specified using composite, parameterised roles. They describe the interaction patterns that a component can engage in. There is one import and one export interaction protocol for each component.

  Example 9 The provided `DocStorageServer` component is based on[4]

  $$\forall \texttt{create} \circ \texttt{id}; !(\texttt{retrieve} \circ \texttt{id} \\ + \texttt{update} \circ (\texttt{id}, \texttt{doc})).\texttt{post}$$

  as the export interaction protocol. □

Our ontological language allows us to specify both safety and liveness properties of components using value restriction and existential quantification, respectively.

Example 10 We can express that eventually (liveness) after executing *create* (safety), a document is deleted:

$$(\forall \texttt{preCond}.\texttt{true}) \sqcap (\forall \texttt{create}.\exists \texttt{delete} \\ .\forall \texttt{postCond}.\texttt{true})$$

which combines safety and liveness properties[5]. □

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. Selected properties of quantified descriptions are:

1. $\forall R.\forall S.C \Leftrightarrow \forall R\,;S.C$
2. $\forall R.C \sqcap D \Leftrightarrow \forall R.C \sqcap \forall R.D$
3. $\forall R \sqcup S.C \Leftrightarrow \forall R.C \sqcup \forall S.C$

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

[5] This corresponds to a dynamic logic formula `[create(id)]` $\langle$`delete(id)`$\rangle$ `true` with precondition `true` combining safety ($[\ldots]\phi$) and liveness ($\langle\ldots\rangle\psi$) properties [9].

Example 11 $\forall$`create;update.postCond` is equivalent to $\forall$`create.`$\forall$`update.postCond`, which allows us to convert role expressions into logical representations. □

We can apply a modal reasoning style here, e.g.

$$\forall \texttt{update} \circ (\texttt{id,doc}).\forall \texttt{postCond}. \\ \texttt{equal(retrieve(id),doc)}$$

corresponds to a (modal) dynamic logic formula

`[update(id,doc)] retrieve(id)=doc`.

4 An ontology for component matching

The two problems that we are concerned with are component description and component matching. In addition to terminological aspects to support component description, ontologies based on description logics also introduce an inference and reasoning framework. Key constructs of description logics to support matching and composition are equivalence and subsumption. In this section, we look at component matching based on contracts including operation behaviour and interaction protocols and how it relates to subsumption reasoning.

4.1 Subsumption – satisfaction and matching

Subsumption is a relationship defined by subset inclusions for concepts and roles.

– 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 [7]. Subsumption can be further characterised by axioms such as the following for concepts C_1 and C_2: $C_1 \sqcap C_2 \sqsubseteq C_1$ or $C_2 \rightarrow C_1$ implies $C_2 \sqsubseteq C_1$.

We use subsumption to reason about matching of two component descriptions based on transitional roles. A variant of subsumption is our tool to express a notion of satisfaction to define matching, essentially capturing *refinement* and *simulation* ideas.

The tractability of reasoning is a central issue for description logics. The richness of our description logic with complex roles that represent interaction protocols and operation parameters has some potentially negative implications for the complexity of reasoning. However, some aspects help to reduce the complexity. We can, for

instance, restrict roles to functional roles. Another beneficial factor is that for composite roles negation is not required. We do not investigate these aspects in depth – most of them have been investigated in detail [7] – only one issue shall be addressed.

A crucial problem is the decidability of the specification if concrete domains are added. Admissible domains guarantee decidability. 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. We can show that our chosen concrete domains – documents and identifiers, see Example 5 – are admissible [5].

4.2 Matching of component operation descriptions

Subsumption is the central reasoning concept of description logics. We now integrate matching of provided and required operation descriptions with this concept.

An operation is functionally specified through pre- and postconditions. Matching of operations is defined in terms of implications on pre- and postconditions and signature matching based on the widely accepted design-by-contract approach [11]. The 'consequence' inference rule, found in dynamic logic [9], describes the *refinement* of operations by weakening preconditions and strengthening postconditions. A matching definition for operations shall be derived from this rule.

A provided operation P **refines** a requested operation R, or P **matches** R, if, firstly,

$$\frac{\forall inSign.in_R \sqcap \forall R.\forall outSign.out_R}{\forall inSign.in_P \sqcap \forall P.\forall outSign.out_P} \left\langle \begin{array}{l} in_P \equiv in_R \wedge \\ out_P \equiv out_R \end{array} \right.$$

(signatures are compatible if the types of corresponding parameters are the same) and, secondly,

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

(a requested operation precondition is weakened and the postcondition is strengthened)[6].

Matching of operation descriptions is a form of refinement. This contravariant inference rule captures matching based on abstract functional behaviour specifications.

Example 12 The provided operation `updDoc` of the document server, see Figs. 2, 4, matches the `update` requirements. Signatures are compatible. Operation `updDoc` has a weaker, less restricted precondition (we assume `valid(doc)` implies `wellFormed(doc)`) and a stronger, more determining postcondition (`retrieve(id)=doc` ∧ `wellFormed(doc)` implies `retrieve(id)=doc`), i.e. the provided operation satisfies the requirements. □

[6] The matching rule defined here is sound, see [5].

Matching implies subsumption, but is not the same. Refinement, i.e. matching of component operations, is a sufficient criterion for subsumption (see [5] for details):

If operation P refines (matches) R, then $P \sqsubseteq R$.

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

This refinement-based definition provides matching foundations within a description logic framework. To support a search engine or a directory service, these foundations would need to be extended. The signature notion can be expanded to include subsignatures or polymorphic signature matching [10]. Pre- and postcondition-based matching can be realised as part of the design-by-contract approach [11].

4.3 Matching of component interaction protocols

Together with operation matching based on functional descriptions, interaction protocol matching is the basis of component matching. Both client and provider components participate in interaction processes based on the operations 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 operations. The provider on the other hand will impose a constraint on the ordering of the execution of operations that are provided through the interaction protocol specification.

A notion of consistency of composite roles for interaction protocols relates to the underlying functional operation specifications based on pre- and postconditions.

- 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.
- A composite role $P(R_1, \ldots, R_n)$ is **consistent**, if the last state of the P execution is reachable.

A composite role P is **consistent** if the following sufficient conditions are satisfied:

1. For each sequence $R;S$ in P:
$$\forall postCond.post_R \sqsubseteq \forall preCond.pre_S$$
2. For each iteration $!R$ in P:
$$\forall postCond.post_R \sqsubseteq \forall preCond.pre_R$$
3. For each choice $R+S$ in P:
$$\forall preCond.pre_R \sqcap \forall preCond.pre_S \text{ and } \forall postCond.post_R \sqcap \forall postCond.post_S$$

A **component interaction protocol** is a consistent composite role $P(R_1,\ldots,R_n)$ constructed from transitional roles and connectors ';' , '!' , and '+'. Interaction protocols are interpreted by **transition graphs** for composite transitional roles, i.e. graphs on states and transitions that represent all possible protocol executions.

An interaction protocol describes the ordering of observable activities of a component. Process calculi suggest simulations and bisimulations as constructs to address the equivalence of interaction protocols. We use a notion of simulation between protocols to define interaction protocol matching between requestor and provider.

A provider interaction protocol $P(S_1,\ldots,S_k)$ **simulates** a requested interaction protocol $R(T_1,\ldots,T_l)$, or protocol 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.

Note, that this simulation subsumes operation matching through the refinement condition at the end. The provider component needs to be able to simulate the request, i.e. needs to meet the expected interaction protocol of the requestor.

Example 13 The provided document server component requires an interaction pattern[7]

```
crtDoc;!(rtrDoc+updDoc);delDoc
```

and the requestor component expects

```
create;!(retrieve+update)
```

as the ordering of output interactions. Assuming that the pairs of operations `crtDoc` and `create`, `rtrDoc` and `retrieve`, and `updDoc` and `update`, respectively, match based on their individual operation behaviour according to the matching definition from Sect. 4.2, the provider matches (simulates) the required server interaction protocol. Service `delDoc` is not requested. □

The simulation definition implies that the association between basic roles (operations) S_i and T_j in two interaction protocols is not fixed, i.e. any S_i such that S_i refines T_j for a requested operation T_j is suitable. For a given T_j, in principle several different provider operations S_i can provide the actual operation execution during the execution process.

As for operation matching, interaction protocol matching is not the same as subsumption. Subsumption on roles is input/output-oriented, whereas simulation needs to consider internal states of composite role executions. For each request in a protocol, there needs to be a corresponding provided operation. However, matching is again a sufficient condition for subsumption:

If the interaction protocol $P(S_1,\ldots,S_k)$ simulates interaction protocol $R(T_1,\ldots,T_l)$, then $R \sqsubseteq P$.

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

Within the service context of the Web, the focus has recently shifted towards service coordination, i.e. composition and process assembly. Consequently, we have extended design-by-contract-based matching from Sect. 4.2 to include interaction protocol matching, providing foundations for a more expressive directory retrieval and composition support. Most directory services are currently based on syntactical matching, with the exception of some service ontologies [4,13].

5 Related work

While various component matching techniques exist – e.g. [10] for matching of polymorphic signatures, [3] for semantics-enhanced matching, and [11] for the design-by-contract method – our aim has been to lay the foundations for these aspects in an ontological framework.

Some effort has already been made to exploit ontology technology for the software domain [4,13]. These approaches have so far focused on individual Web services. Service ontologies add non-functional properties into description and matching – an approach that has also been looked at for CBSE, see [14]. OWL-S [4] (previously called DAML-S) is an OWL ontology for describing properties of Web services. OWL-S represents services as concepts. We, in contrast, represent component operations as roles and not as concepts, giving a more process-oriented focus. Component behaviour and processes have been recognised as central aspects. In [12], a framework similar to ours, based on a process calculus interpreted in transition systems, is introduced. While our focus is on proces-oriented match-

[7] We drop parameters in protocol expressions for illustration, if, as in this case, only the ordering is relevant.

ing, theirs is a complementary approach on deadlock and other analyses.

OWL-S [4] relies on OWL subsumption reasoning to match requested and provided Web services. OWL-S provides to some extent for Web services what we aim at for components. However, the form of reasoning and ontology support that we provide here is not possible in OWL-S, since services are modelled as concepts and not rules. Only considering services as roles would make modal reasoning about component behaviour possible.

Schild [8] 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 [9] into description logics. We have expanded Schild's results by representing names in the notation and by defining a modal logic-influenced matching inference framework in a knowledge representation setting. A few knowledge representation issues, however, can be addressed in the future in order to enhance the description logic developed here [7]. Assertions about data types can also be represented as intentional knowledge. Epistemic operators have been introduced for this purpose.

6 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 emphasise here the importance of formalising central development activities, such as component matching in form of ontologies. In the Web context, service and component technologies are moving towards each other. Web services exhibit component character in the assembly of service-oriented architectures from reusable service components.

Our overall objective has been to provide reasoning support for semantically described components. We have presented a 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 operations requested or provided in these interactions. The reasoning capabilities that we have obtained and represented in form of a matching ontology go beyond current ontologies for service or component matching. Even though description logics have been developed to address knowledge representation problems in general, a connection to modal logics has allowed us to obtain a rich framework for representing and reasoning about components. Description logic is central for various reasons. Firstly, it is a framework focusing strongly on the tractability of reasoning; secondly, it is suitable for the integration of component technology into the Web environment and its standards; and, thirdly, it allows other knowledge engineering techniques, such as domain modelling, to be integrated.

References

1. Szyperski, C: Component Software: Beyond Object-Oriented Programming, 2nd edn. Addison-Wesley, Reading (2002)
2. Leavens, G.T., Sitamaran, M.: Foundations of Component-Based Systems. Cambridge University Press, Cambridge (2000)
3. Moorman Zaremski, A., Wing, J.M.: Specification matching of software components. ACM Trans. Softw. Eng. Methods, **6**(4), 333–369 (1997)
4. DAML-S Coalition. DAML-S: Web Services Description for the Semantic Web. In: Horrocks, I., Hendler, J. (eds.) Proccedings of the First International Semantic Web Conference ISWC 2002, pp. 279–291. LNCS 2342, Springer, Berlin Heidelbery New York (2002)
5. Pahl, C.: An ontology for software component matching. In: Pezzè, M. (eds.) Proccedings of the Fundamental Approaches to Software Engineering FASE'2003, pp. 6–21. LNCS 2621, Springer, Berlin Heidelbery New York (2003)
6. W3C Semantic Web Activity: Semantic Web Activity Statement. http://www.w3.org/2001/sw. (visited 06/12/2004) 2004
7. Baader, F., McGuiness, D., Nardi, D., Schneider, P.P. (eds.): The Description Logic Handbook. Cambridge University Press, Cambridge (2003)
8. Schild, K.: A Correspondence Theory for Terminological Logics: Preliminary Report. In: Proccedings of the 12th International Joint Conference on Artificial Intelligence (1991)
9. Kozen, D., Tiuryn, J.: Logics of programs. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science, vol. B, pp. 789–840. Elsevier, Amsterdam (1990)
10. Gastinger, S., Hennicker, R., Stabl, R.: Design of Modular Software Systems with Reuse. In: Broy, M., Jähnichen, S. (eds.) KORSO – Methods, Languages, and Tools for the Construction of Correct Software, pp. 112–127. LNCS 1009, Springer, Berlin Heidelberg New York (1995)
11. Meyer, B.: Applying design by contract. Computer **25**(10), 40–51 (1992)
12. Inverardi, P., Tivoli, M.: Software Architecture for Correct Components Assembly. In: Formal Methods for the Design of Computer, Communication and Software Systems: Software Architecture. LNCS Series, Springer, Berlin Heidelberg New York (2003)
13. Lara, R., Roman, D., Polleres, A., Fensel, D.: A Conceptual Comparison of WSMO and OWL-S. In: Zhang, L.-J., Jeckle, M. (eds.) European Conference on Web Services ECOWS 2004, pp. 254–269, LNCS 3250, Springer, Berlin Heidelberg New York (2004)
14. Reussner, R., Poernomo, I., Schmidt, H.: Contracts and quality attributes for software components. In: Weck, W., Bosch, J., Szyperski, C. (eds.) Proccedings of the 8th International Workshop on Component-Oriented Programming WCOP'03 (2003)