# Organising the Knowledge Space for Software Components

Claus Pahl

School of Computing,
Dublin City University, Dublin 9, Ireland

**Abstract.** Software development has become a distributed, collaborative process based on the assembly of off-the-shelf and purpose-built components. The selection of software components from component repositories and the development of components for these repositories requires an accessible information infrastructure that allows the description and comparison of these components.

General knowledge relating to software development is equally important in this context as knowledge concerning the application domain of the software. Both form two pillars on which the structural and behavioural properties of software components can be expressed. Form, effect, and intention are the essential aspects of process-based knowledge representation with behaviour as a primary property.

We investigate how this information space for software components can be organised in order to facilitate the required taxonomy, thesaurus, conceptual model, and logical framework functions. Focal point is an axiomatised ontology that, in addition to the usual static view on knowledge, also intrinsically addresses the dynamics, i.e. the behaviour of software. Modal logics are central here – providing a bridge between classical (static) knowledge representation approaches and behaviour and process description and classification.

We relate our discussion to the Web context, looking at Web services as components and the Semantic Web as the knowledge representation framework.

## 1 Introduction

The style of software development has changed dramatically over the past decades. Software development has become a distributed, collaborative process based on the assembly of off-the-shelf and purpose-built software components – an evolutionary process that in the last years has been strongly influenced by the Web as a software development and deployment platform.

This change in the development style has an impact on information and knowledge infrastructures surrounding these software components. The selection of components from component repositories and the development of components for these repositories requires an accessible information infrastructure that allows component description, classification, and comparison. Organising the space of knowledge that captures the description of properties and the classification of software components based on these descriptions is central. Discovery and composition of software components based on these

descriptions and classifications have become central activities in the software development process (Crnkovic and Larsson (2002)). In a distributed environment where providers and users of software components meet in electronic marketplaces, knowledge about these components and their properties is essential; a shared knowledge representation language is a prerequisite (Horrocks et al. (2003)). Describing software behaviour, i.e. the effect of the execution of services that a component might offer, is required.

We will introduce an ontological framework for the description and classification of software components that supports the discovery and composition of these components and their services – based on a formal, logical coverage of this topic in (Pahl (2003)). Terminology and logic are the cornerstones of our framework. Our objective is here twofold:

- We will illustrate an ontology based on description logics (a logic underlying various ontology languages), i.e. a logic-based terminological classification framework based on (Pahl (2003)). We exploit a connection to modal logics to address behavioural aspects, in particular the safety and liveness of software systems.
- Since the World-Wide Web has the potential of becoming central in future software development approaches, we investigate whether the Web can provide a suitable environment for software development and what the requirements for knowledge-related aspects are. In particular Semantic Web technologies are important for this context.

We approach the topic here from a general knowledge representation and organisation view, rather than from a more formal, logical perspective.

In Section 2 we describe the software development process in distributed environments in more detail. In Section 3, we relate knowledge representation to the software development context. We define an ontological framework for software component description, supporting discovery and composition, in Section 4. We end with some conclusions in Section 5.

## 2 The software development process

The World-Wide Web is currently undergoing a change from a document- to a services-oriented environment. The vision behind the Web Services Framework is to provide an infrastructure of languages, protocols, and tools to enable the development of services-oriented software architectures on and for the Web (W3C (2004)). Service examples range from simple information providers, such as weather or stock market information, to data storage support and complex components supporting e-commerce or online banking systems. An example for the latter is an account management component offering balance and transfer services. Service providers advertise their services; users (potential clients of the provider) can browse repository-based marketplaces to find suitable services, see Fig. 1. The prerequisite is a common

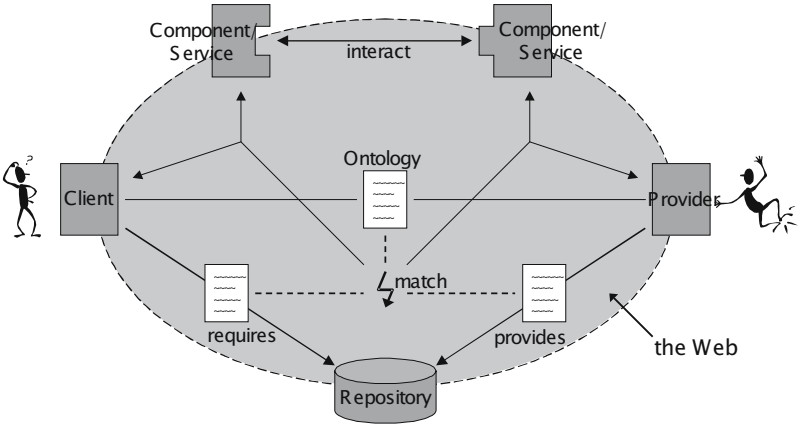


**Fig. 1.** A Service Component Development Scenario.

language to express properties of these Web-based services and a classification approach to organise these. The more knowledge is available about these services, the better can a potential client determine the suitability of an offer.

Services and components are related concepts. Web services can be provided by software components; we will talk about service components in this case. If services exhibit component character, i.e. are self-contained with clearly defined interfaces that allows them to be reused and composed, then their composition to larger software system architectures is possible. Pluggable and reusable software components are one of the approaches to software developments that promises risk minimisation and cost reduction. Composition can be physical, i.e. a more complex artefact is created through assembly, or logical, i.e. a complex system is created by allowing physically distributed components to interact. Even though our main focus are components in general, we will discuss them here in the context of the Web Services platform.

The ontological description of component properties is our central concern (Fig. 1). We will look at how these descriptions are used in the software development process. Two activities are most important:

- Discovery of provided components (lower half of Fig. 1) in structured repositories. Finding suitable, reusable components for a given development based on abstract descriptions is the problem.
- Composition of discovered components in complex service-based component architectures through interaction (upper half of Fig. 1). Techniques are needed to compose the components in a consistent way based on their descriptions.

For a software developer, the Web architecture means that most software development and deployment activities will take place outside the boundaries of her/his own organisation. Component descriptions can be found in external

repositories. These components might even reside as provided services outside the own organisation. Shared knowledge and knowledge formats become consequently essential.

## 3 A knowledge space for software development

The Web as a software platform is characterised by different actors, different locations, different organisations, and different systems participating in the development and deployment of software. As a consequence of this heterogeneous architecture and the development paradigm as represented in Fig. 1, shared and structured knowledge about components plays a central role. A common understanding and agreement between the different actors in the development process are necessary.

A shared, organised *knowledge space* for software components in service-oriented architectures is needed. The question how to organise this knowledge space is the central question of this paper. In order to organise the knowledge space through an ontological framework (which we understand essentially as basic notions, a language, and reasoning techniques for sharable knowledge representation), we address three *facets of the knowledge space*: firstly, types of knowledge that is concerned, secondly, functions of the knowledge space, and, finally, the representation of knowledge (Sowa (2000)).

Three *types of knowledge* can be represented in three layers:

- The *application domain* as the basic layer.
- Static and dynamic *component properties* as the central layer.
- Meta-level *activity-related knowledge* about discovery and composition.

We distinguish four *knowledge space functions* (Daconta et al. (2003)) that characterise how knowledge is used to support the development activities:

- *Taxonomy* – terminology and classification; supporting structuring and search.
- *Thesaurus* – terms and their relationships; supporting a shared, controlled vocabulary.
- *Conceptual model* – a formal model of concepts and their relationships; here of the application domain and the software technology context.
- *Logical theory* – logic-supported inference and proof; here applied to behavioural properties.

The third facet deals with how knowledge is represented. In general, *knowledge representation* (Sowa (2000)) is concerned with the description of entities in order to define and classify these. Entities can be distinguished into objects (static entities) and processes (dynamic entities). *Processes* are often described in three *aspects* or *tiers*:

- *Form* – algorithms and implementation – the 'how' of process description

- *Effect* – abstract behaviour and results – the 'what' of process description
- *Intention* – goal and purpose – the 'why' of process description

We have related the aspects form, effect, and intention to software characteristics such as algorithms and abstract behaviour. The service components are software entities that have process character, i.e. we will use this three-tiered approach for their description.

The three facets of the knowledge space outline its structure. They serve as requirements for concrete description and classification techniques, which we will investigate in the remainder.

## 4 Organising the knowledge space

### 4.1 Ontologies

Ontologies are means of knowledge representation, defining so-called shared conceptualisations. Ontology languages provide a notation for terminological definitions that can be used to organise and classify concepts in a domain. Combined with a symbolic logic, we obtain a framework for specification, classification, and reasoning in an application domain. Terminological logics such as description logics (Baader et al. (2003)) are an example of the latter.

The Semantic Web is an initiative for the Web that builds up on ontology technology (Berners-Lee et al. (2001)). XML – the eXtensible Markup Language – is the syntactical format. RDF – the Resource Description Framework – is a triple-based formalism (subject, property, object) to describe entities. OWL – the Web Ontology Language – provides additional logic-based reasoning based on RDF.

We use Semantic Web-based ontology concepts to formalise and axiomatise processes, i.e. to make statements about processes and to reason about them. Description logic, which is used to define OWL, is based on concept and role descriptions (Baader et al. (2003)). *Concepts* represent classes of objects; *roles* represent relationships between concepts; and *individuals* are named objects. Concept descriptions are based on primitive logical combinators (negation, conjunction) and hybrid combinators (universal and existential quantification). Expressions of a description logic are interpreted through sets (concepts) and relations (roles).

We use a connection between *description logic* and *dynamic logic* (Sattler et al. (2003), Chapter 4.2.2). A dynamic logic is a modal logic for the description of programs and processes based on operators to express necessity and possibility (Kozen and Tiuryn (1990)). This connection allows us to address safety (necessity of behaviour) and liveness (possibility of behaviour) aspects of service component behaviour by mapping the two modal operators 'box' (or 'always', for safety) and 'diamond' (or 'eventually', for liveness) to the description logic universal and existential quantification, respectively. The central idea behind this connection is that roles can be interpreted as

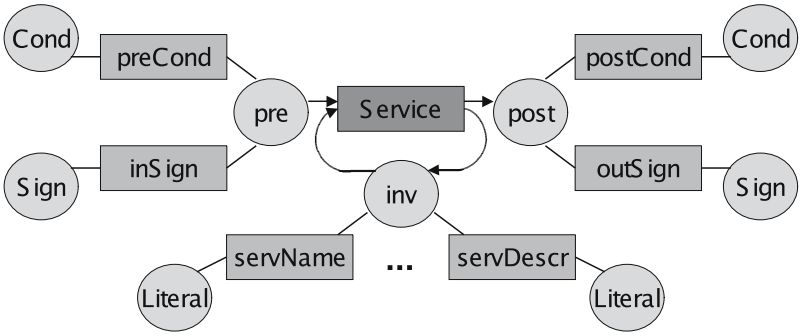


**Fig. 2.** A Service Component Ontology.

accessibility relations between states, which are central concepts of process-oriented software systems. The correspondence between description logics and a multi-modal dynamic logic is investigated in detail in (Schild (1991)).

### 4.2 A discovery and composition ontology

An intuitive approach to represent software behaviour in an ontological form would most likely be to consider components or services as the central concepts (DAML-S Coalition (2002)). We, however, propose a different approach. Our objectve is to represent software systems. These systems are based on inherent notions of state and state transition. Both notions are central in our approach. Fig. 2 illustrates the central ideas. Service executions lead from old (pre)states to new (post)states, i.e. the service is represented as a role (a rectangle in the diagram), indicated through arrows. The modal specifications characterise in which state executions might (using the possibility operator to express liveness properties) or should (using the necessity operator to express safety properties) end. For instance, we could specify that a customer may (possibly) check his/her account balance, or, that a transfer of money must (necessarily) result in a reduction of the source account balance. Transitional roles such as *Service* in Fig. 2 are complemented by more static, descriptional roles such as *preCond* or *inSign*, which are associated through non-directed connections. For instance, *preCond* associates a precondition to a prestate; *inSign* associates the type signatures of possible service parameters. Some properties, such as the service name *servName*, will remain invariant with respect to state change.

Central to our approach is the intrinsic specification of process behaviour in the ontology language itself. Behaviour specifications based on the descriptions of necessity and possibility are directly accessible to logic-based methods. This makes reasoning about behaviour of components possible.

We propose a *two-layered ontology* for discovery and composition. The *upper ontology layer* supports *discovery*, i.e. addresses description, search,

discovery, and selection. The *lower ontology layer* supports *composition*, i.e. addresses the assembly of components and the choreography of their interactions. We assume that execution-related aspects are an issue of the provider – shareable knowledge is therefore not required.

Table 1 summarises development activities and knowledge space aspects. It relates the activities discovery, composition, and execution on services (with the corresponding ontologies) to the three knowledge space facets.

**Table 1.** Development Activities and Knowledge Space Facets.

| | Knowledge Aspect | Knowledge Type | Function |
|---|---|---|---|
| Discovery (upper ontology) | intention (terminology) | domain | taxonomy thesaurus |
| Composition (lower ontology) | effect (behaviour) | component component activities | conceptual model logical theory |
| Execution | form (implementation) | component | conceptual model |

### 4.3 Description of components

Knowledge describing software components is represented in three layers. We use two ontological layers here to support the abstract properties.

- The *intention* is expressed through assumptions and goals of services in the context of the application domain.
- The *effect* is a contract-based specification of system invariants, pre- and postconditions describing the obligations of users and providers.
- The *form* defines the implementation of service components, usually in a non-ontological, hidden format.

We focus on effect descriptions here. Effect descriptions are based on modal operators. These allow us to describe process behaviour and composition based on the choreography of component interactions. The notion of composition shall be clarified now. Composition in Web- and other service-oriented environments is achieved in a logical form. Components are provided in form of services that will reside in their provider location. Larger systems are created by allowing components to interact through remote operation invocation. Components are considered as independent concurrent processes that can interact (communicate) with each other. Central in the composition are the abstract effect of individual services and the interaction patterns of components as a whole.

We introduce role expressions based on the role constructors sequential composition $R;S$, iteration $!R$, and choice $R+S$ into a basic ontology language to describe interaction processes (Pahl (2003)). We often use $R \circ S$

instead of $R;S$ if $R$ and $S$ are functional roles, i.e. are interpreted by functions – this notation will become clearer when we introduce names and service parameters. Using this language, we can express ordering constraints for parameterised service components. These process expressions constrain the possible interaction of a service component with a client.

For instance, $Login;!(Balance + Transfer)$ is a role expression describing an interaction process of an online banking user starting with a login, then repeatedly executing balance enquiry or money transfer.

An effect specification[1] focussing on safety is for a given system state

$$\forall preCond.positive(Balance(no)) \quad \text{and}$$
$$\forall Transfer.\forall postCond.reduced(Balance(no))$$

saying that if the account balance for account *no* is positive, then money can be transfered, resulting (necessarily) in a reduced balance. *Transfer* is a service; $positive(Balance(no))$ and $reduced(Balance(no))$ are pre- and postcondition, respectively. These conditions are concept expressions. The specification above is formed by navigating along the links created by roles between the concepts in Fig. 2 – *Transfer* replaces *Service* in the diagram.

In Fig. 3, we have illustrated two sample component descriptions – one representing the requirements of a (potential) client, the other representing a provided bank account component. Each component lists a number of individual services (operations) such as *Login* or *Balance*. We have used pseudocode for signatures (parameter names and types) and pre-/postconditions – a formulation in proper description logic will be discussed later on. We have limited the specification in terms of pre- and postconditions to one service, *Transfer*.

The requirements specification forms a query as a request, see Fig. 1. The ontology language is the query language. The composition ontology provides the vocabulary for the query. A query should result ideally in the identification of a suitable (i.e. matching) description of a provided component. In our example, the names correspond – this, however, is in general not a matching prerequisite. Behaviour is the only definitive criterion.

### 4.4 Discovery and composition of components

Component-based development is concerned with discovery and composition. In the Web context, both activities are supported by Semantic Web and Web Services techniques. They support semantical descriptions of components, marketplaces for the discovery of components based on *intention* descriptions as the search criteria, and composition support based on semantic *effect* descriptions. The deployment of components is based on the *form* description.

[1] This safety specification serves to illustrate effect specification. We will improve this currently insufficient specification (negative account balances are possible, but might not be desired) in the next section when we introduce names and parameters.

**Component** `AccountRequirements`
*signatures and pre-/postconditions*
`Login`
*inSign* `no:int,user:string`
*outSign* `void`
`Balance`
*inSign* `no:int`
*outSign* `real`
`Transfer`
*inSign* `no:int,dest:int,sum:real`
*outSign* `void`
*preCond* `Balance(no) ≥ sum`
*postCond* `Balance(no) = Balance(no)@pre - sum`
`Logout`
*inSign* `no:int`
*outSign* `void`
*interaction process*
`Login;!Balance;Logout`

**Component** `BankAccount`
*signatures and pre-/postconditions*
`Login(no:int,user:string)`
*inSign* `no:int,user:string`
*outSign* `void`
`Balance(no:int):real`
*inSign* `no:int`
*outSign* `real`
`Transfer(no:int,dest:int,sum:real)`
*inSign* `no:int,dest:int,sum:real`
*outSign* `void`
*preCond* `true`
*postCond* `Balance(no) = Balance(no)pre - sum`
`Logout(no:int)`
*inSign* `no:int`
*outSign* `void`
*interaction process*
`Login;!(Balance+Transfer);Logout`

**Fig. 3.** Bank Account Component Service.

**Query and Discovery.** The aim of the discovery support is to find suitable provided components in a first step that match based on the application domain related goals and that, in a second step, match based on the more technical effect descriptions. Essentially, the ontology language provides a query language. The client specifies the requirements in a repository query in terms of the ontology, which have to be matched by a description of a provided component.

Matching requires technical support, in particular for the formal effect descriptions. Matching can be based on techniques widely used in software development, such as refinement (which is for instance formalised as the consequence notion in dynamic logic). We will focus on the description of effects, i.e. the lower ontology layer (cf. Fig. 2):

- Service component-based software systems are based on a central state concept; additional concepts for auxilliary aspects such as the pre- and poststate-related descriptions are available.
- Service components are behaviourally characterised by transitional roles (for state changes between prestate and poststate) and descriptional roles (auxilliary state descriptions).

**Matching and composition.** In order to support matching and composition of components through ontology technology, we need to extend the (already process-oriented) ontology language we presented above (Pahl and Casey (2003)). We can make statements about component interaction processes, but we cannot refer to the data elements processed by services. The role expression sublanguage needs to be extended by names (representing data elements) and parameters (which are names passed on to services for processing):

- Names: a name is a role $n[Name]$ defined by the identity relation on the interpretation of an individual $n$.
- Parameters: a parameterised role is a transitional role $R$ applied to a name $n[Name]$, i.e. $R \circ n[Name]$.

We can make our *Transfer* service description more precise by using a data variable ($sum$) in pre- and postconditions and as a parameter:

$\forall preCond.(Balance(no) \geq sum)$ and
$\forall Transfer \circ sum[Name].\forall postCond.(Balance(no) = Balance(no)@pre - sum)$

This specification requires *Transfer* to decrease the pre-execution balance by $sum$.

Matching needs to be supported by a comparison construct. We already mentioned a refinement notion as a suitable solution. This definition, however, needs to be based on the support available in description logics. Subsumption is the central inference technique. Subsumption is the subclass relationship on concept and role interpretations. We define two types of *matching*:

- For *individual services*, we define a *refinement* notion based on weaker preconditions (allowing a service to be invoked in more states) and stronger postconditions (improving the results of a service execution). For example *true* as the precondition and $Balance(no) = Balance(no)@pre - sum$ as the postcondition for $Transfer \circ sum[Name]$ matches, i.e. refines the requirements specification with $Balance(no) >= sum$ as the precondition

and $Balance(no) = Balance(no)@pre - sum$ as the postcondition since it allows the balance to become negative (i.e. allows more flexibility for an account holder).

- For *service processes*, we define a *simulation* notion based on sequential process behaviour. A process matches another process if it can simulate the other's behaviour. For example the expression $Login; !(Balance + Transfer); Logout$ matches, i.e. simulates $Login; !Balance; Logout$, since the transfer service can be omitted.

Both forms of matching are sufficient criteria for subsumption. Matching of effect descriptions is the prerequisite for the composition of services. Matching guarantees the proper interaction between composed service components.

## 5 Conclusions

Knowledge plays an important role in the context of component- and service-oriented software development. The emergence of the Web as a development and deployment platform for software emphasises this aspect.

We have structured a knowledge space for software components in service-oriented architectures. Processes and their behavioural properties were the primary aspects. We have developed a process-oriented ontological model based on the facets form, effect, and intention. The discovery and the composition of process-oriented service components are the central activities. This knowledge space is based on an ontological framework formulated in a description logic. The defined knowledge space supports a number of different functions – taxonomy, thesaurus, conceptual model, and logical theory. These functions support a software development and deployment style suitable for the Web and Internet environment.

Explicit, machine-processable knowledge is the key to future automation of software development activities. In particular, Web ontologies have the potential to become an accepted format that supports such an automation endeavour.

**Acknowledgements**

The author is greatly indebted to the anonymous reviewers for their helpful comments and suggestions.

## References

BAADER, F., MCGUINESS, D., NARDI, D. and SCHNEIDER, P. (Eds.) (2003): *The Description Logic Handbook*. Cambridge University Press.

BERNERS-LEE, T., HENDLER, J. and LASSILA, O. (2001): The Semantic Web. *Scientific American, 284(5)*.

CRNKOVIC, I. and LARSSON, M. (Eds.) (2002): *Building Reliable Component-based Software Systems*. Artech House Publishers.

DACONTA, M.C., OBRST, L.J. and SMITH, K.T. (2003): *The Semantic Web – A Guide to the Future of XML, Web Services, and Knowledge Management*. Wiley & Sons.

DAML-S COALITION (2002): DAML-S: Web Services Description for the Semantic Web. In: I. Horrocks and J. Hendler (Eds.): *Proc. First International Semantic Web Conference ISWC 2002*. Springer-Verlag, Berlin, 279–291.

HORROCKS, I., MCGUINESS, D. and WELTY, C. (2003): Digital Libraries and Web-based Information Systems. F. Baader, D. McGuiness, D. Nardi and P. Schneider (Eds.): *The Description Logic Handbook*. Cambridge University Press.

KOZEN, D. and TIURYN, J. (1990): Logics of programs. In: J. van Leeuwen (Ed.): *Handbook of Theoretical Computer Science, Vol. B*. Elsevier Science Publishers, 789–840.

PAHL, C. (2003): An Ontology for Software Component Matching. In: *Proc. Fundamental Approaches to Software Engineering FASE'2003*. Springer-Verlag, Berlin, 208–216.

PAHL, C. and CASEY, M. (2003): Ontology Support for Web Service Processes. In: *Proc. European Software Engineering Conference / Foundations of Software Engineering ESEC/FSE'03*. ACM Press.

SATTLER, U., CALVANESE, D. and MOLITOR, R. (2003): Description Logic - Relationships with other Formalisms. In: F. Baader, D. McGuiness, D. Nardi and P. Schneider (Eds.): *The Description Logic Handbook*. Cambridge University Press.

SOWA, J.F. (2000): *Knowledge Representation – Logical, Philosophical, and Computational Foundations*. Brooks/Cole.

SCHILD, K. (1991): A Correspondence Theory for Terminological Logics: Preliminary Report. In *Proc. 12th Int. Joint Conference on Artificial Intelligence*.

W3C – WORLD WIDE WEB CONSORTIUM (2004): *Web Services Framework*. http://www.w3.org/2002/ws.