Designing Component-Based Semantic Web Applications with DESWAP

Olaf Hartig
Humboldt-Universität zu Berlin
Institut für Informatik
hartig@informatik.hu-berlin.de

Martin Kost
Humboldt-Universität zu Berlin
Institut für Informatik
kost@informatik.hu-berlin.de

Johann-Christoph Freytag
Humboldt-Universität zu Berlin
Institut für Informatik
freytag@informatik.hu-berlin.de

ABSTRACT

We present the DESWAP[1] system that relieves developers of component-based Semantic Web applications of the burden of manual component selection (CS). Our system implements a novel approach to automatic CS that utilizes semantic technologies. We enable users to specify dependencies between the required components, an issue not considered by existing approaches. To realize our approach in DESWAP we developed a knowledge base with comprehensive semantic descriptions of software and their functionalities.

1. INTRODUCTION

A component-based software system realizes specific tasks by the integration of existing software components; the components offer functionalities necessary to implement these tasks. Due to the reuse of software, component-based development (CBD) promises reduced development times, increased flexibility, and increased reliability [2].

The main challenge of CBD is finding and selecting components, often denoted as the *component selection* (CS) problem. Finding candidate components for each required functionality may become laborious. From the set of all possible candidate components a subset must be selected which satisfies the developers' objectives. A specific characteristic of these objectives that has not been considered in existing CS approaches such as [1], [3], and [4], are dependencies between functionalities. Usually, the functionalities that may be realized by components rely on each other. For instance, a system may store the result of a remote query to a database; since the data import format must be compatible with the format of the query result the storage functionality depends on the query functionality. These kinds of dependencies add a higher degree of complexity to the selection process: selecting a component for one of the functionalities reduces the candidates for dependent functionalities to components that are compatible with the selected one.

Since finding and selecting components will quickly become too complex to be performed manually we developed a framework that supports developers to solve the CS problem automatically. In this paper we present our approach which applies semantic technologies such as ontologies, rules and reasoning. To enable automatic CS we represent the software components as well as the CS-specific requirements in a machine-processable form (cf. Section 2). Based on this representation we developed strategies for machine-based CS (cf. Section 3). We implemented our concepts in the DESWAP system (cf. Section 4) that provides a sophisticated CS tool and a machine- and human-accessible catalog of software components for Semantic Web applications.

[1]Development Environment for Semantic Web APplications

2. OUR DATA MODEL

The essential requirement for automatic CS is a machine-processable representation of the requirements for CS and of the available software. Therefore, we developed a comprehensive ontology of software and requirements.

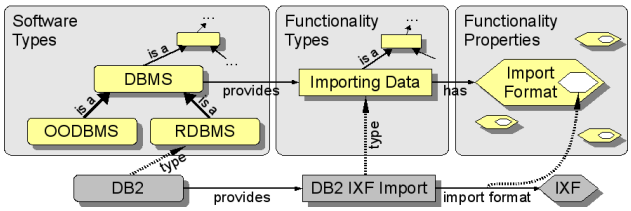

Figure 1: Extract of a software catalog.

Figure 1 illustrates some of the main concepts in our ontology that deal with software. We classify *software* in a hierarchy of *software types*. Each software offers certain *functionalities*. These functionalities are classified in a hierarchy of *functionality types*. All software of the same type offers the same types of functionalities (e.g. each DBMS can import data); hence, we define software types by the types of functionalities they offer. The functionality types are specified by sets of typical properties, called *functionality properties*; each actual functionality that is offered by a specific software has specific values for its functionality properties. For instance, data import functionalities in general support an import format; data import in DB2 in particular supports the IXF format [5]. Besides the aforementioned concepts we model software versions, composed software, dependencies of software, and other properties such as licenses and prices. Our ontology enables the realization of a sophisticated software catalog. Due to the ontology the descriptions in the catalog have a machine-processable meaning; our automatic CS tool (cf. Section 4) utilizes these meanings to discover potential software components.

In addition to software, our ontology represents *CS requirements* which contain required functionalities and dependencies between these functionalities. *Required functionalities* are those functionalities of a component-based system

that may be realized by components. Required functionalities are specified by a type and an optional set of property restrictions. The *type* of a required functionality refers to one of the functionality types that are associated with the software types as mentioned before. Hence, each software that offers functionalities of this type could potentially be selected as a component that realizes the required functionality. However, *property restrictions* limit the set of potential candidates. These restrictions predefine particular values which are permitted for the functionality properties of the corresponding functionality. For instance, a possible restriction for a required data import functionality would be the requirement of IXF as import format. Only those software that offers a functionality with the permitted properties support the required functionality.

Additionally, we represent *dependencies* between required functionalities (cf. Section 1) as a part of the CS requirements. To specify conditions under which functionalities can be combined without conflict we introduce *composition policies*. We use these policies to verify whether a selection of software is compatible with respect to the dependencies between required functionalities. For each pair of functionality types a composition policy identifies those functionality properties that must have mutually compatible values. For instance, the policy for storage functionalities that depend on query functionalities specifies that the storage import format must be compatible with the query result format.

3. FINDING APPROPRIATE SELECTIONS

Based on our representation of software and requirements we developed a method for automatic CS. A *local solution* for a required functionality is a software that offers a functionality which can implement the required functionality. A selection of software that satisfies all CS requirements is a *global solution*. To satisfy all CS requirements a selection must associate every required functionality with a software from its set of local solutions; furthermore, the selection must be compatible with respect to the dependencies between required functionalities, i.e., the functionalities offered by the associated software must not violate the composition policies for the respective dependencies. A naive approach to find a global solution is to iterate over all selections that combine exactly one software from each set of local solutions until a selection is found that does not violate the dependencies. This naive strategy is too inefficent, especially for complex requirements with many dependencies.

Our method reduces the search space by a propagation of property restrictions. For instance, a storage functionality may depend on a query functionality of which the result format is restricted to XML; if the corresponding composition policy demands compatibility for the query result format and the storage import format then the import format is implicitly restricted to XML. We apply our composition policies as rules that propagate property restrictions and, thus, make the implicit restrictions explicit. Since propagation adds further property restrictions to the required functionalities it reduces the sets of local solutions. However, the propagation cannot only be applied to the user-specified property restrictions. Selecting a software from the local solutions of a required functionality for the global solution yields additional restrictions for the respective required functionality. By propagating these restrictions we can reduce the sets of local solutions even further.

We propose an algorithm that constructs a global solution by incrementally adding one candidate from each set of local solutions; with each addition the algorithm propagates the restrictions and reduces the sets of local solutions (cf. Figure 2). If a set of local solutions becomes empty it is impossible to construct a global solution with the selected candidates. In this case our algorithm applies a backtracking strategy to try different candidates. The algorithm terminates when a global solution has been completed or when all combinations of candidates have been considered without avail.

Figure 2: The main steps of our CS method.

We currently do not consider optimality criteria such as a minimal number of software in the selection. However, we are working on an extension of our algorithm that finds optimal global solutions.

4. THE DESWAP SYSTEM

We implemented our concepts in the DESWAP system which is primarily intended to be used for component-based systems that apply Semantic Web technologies. DESWAP provides a sophisticated CS tool and a machine- as well as human-accessible software catalog.

With the software catalog we provide a comprehensive description of software and functionalities that are relevant for Semantic Web applications. In addition to the descriptions we provide composition policies for the functionalities in the catalog. We represent our ontology with OWL; based on our ontology we realize the software catalog as a knowledge base with OWL descriptions. A SPARQL endpoint provides machine-based access to the knowledge base. For human users we provide a Web-based interface that enables browsing as well as editing the data in the catalog.

The CS tool of DESWAP enables the specification of CS requirements and determines a suitable selection of software components which satisfies the requirements. The tool realizes our CS method. To find candidate software that offers required functionalities the tool accesses our software catalog. However, by using a reasoner the tool does not only consider explicit statements about the functionalities of software, but, it evaluates inferred statements and, hence, discovers more potential candidates.

5. REFERENCES

[1] S. E. Carlson. Genetic algorithm attributes for component selection. *Engineering Design*, 8(1), 1996.

[2] P. C. Clements. From subroutines to subsystems: Component-based software development. *The American Programmer*, 11(8), 1995.

[3] M. R. Fox et al. Approximating component selection. In *ACM/IEEE Winter Simulation Conference*, 2004.

[4] N. Haghpanah et al. Approximation algorithms for software component selection problem. In *Proc. of the Asia-Pacific Software Engineering Conference*, 2007.

[5] IBM. *Data Movement Utilities Guide and Reference*. DB2 Version 9.5 Manuals, Mar.2008.

DEMO OF THE DESWAP SYSTEM

In our demonstration we will present the DESWAP system with its three main use cases: designing component-based Semantic Web applications, browsing the software catalog, and updating the software catalog. We implemented the DESWAP system as a JSP-based Web application that uses the Jena[2] framework to accesses the DESWAP knowledge base. The knowledge base itself consists of five OWL documents, an RDF repository, the domain-specific composition policies, and a reasoner; the OWL documents define our ontology (cf. Section 2), the RDF store contains OWL descriptions of the software catalog, and the reasoner, Pellet[3] in our case, discovers implicit knowledge.

With the first use case the DESWAP system supports developers to design component-based Semantic Web applications and to find suitable software components. Our aim was to develop a tool that seamlessly integrates in common software development processes. Developers usually create a model of the software before implementing it; they define the use cases and the activities that realize each use case. Based on our understanding of a component-based system (cf. Section 1) the activities might be implemented by the integration of existing software. Hence, developers must specify which functionalities a software has to offer in order to be integrated. Our system enables developers to specify their CS requirements: they can define the required functionalities for each activity and they can define the dependencies between these functionalities (cf. Figure 3).

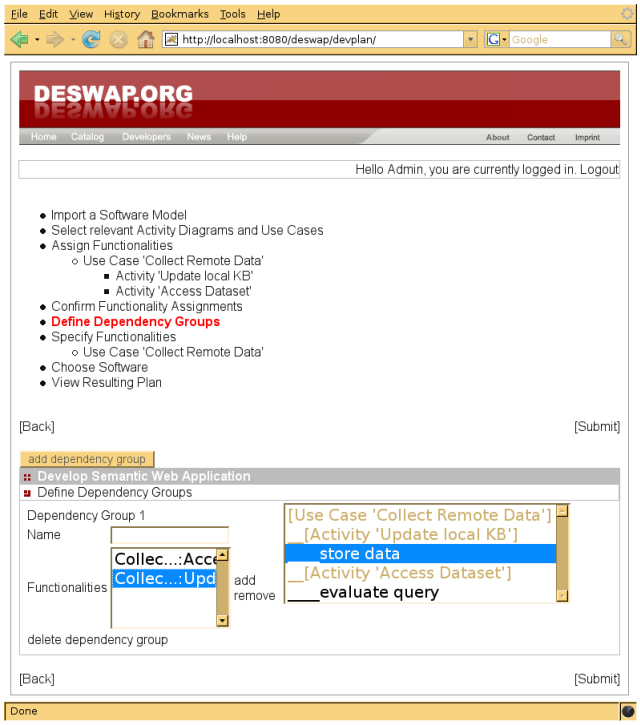

Figure 3: Defining dependencies between required functionalities with DESWAP.

Developers will use their software model to specify the CS requirements. In order to integrate DESWAP in the development process our system imports the software models created by a developer[4]. After supporting the users to specify their CS requirements DESWAP applies our CS strategies (cf. Section 3) and determines a suitable selection of software components which satisfies the requirements.

The second use case allows users to browse the software catalog. While the knowledge base contains comprehensive descriptions of available software and its functionalities the user interface hides most of the complexity of these semantic descriptions. For instance, for the user interface we provide a simplified hierarchy of software types which we generate from the full hierarchy. Users can browse the simple hierarchy to find summaries of the available data about software, its versions, and the functionalities offered by each version (cf. Figure 4). Furthermore, users can rate and review software.

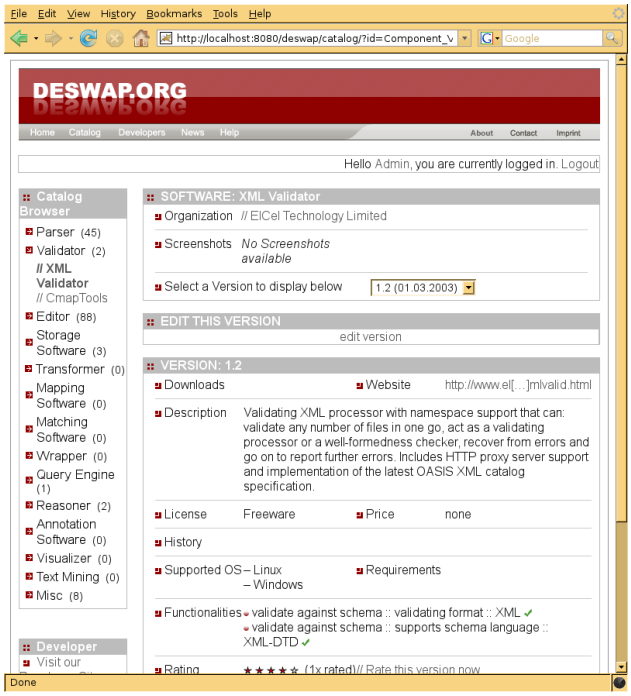

Figure 4: The catalog browser of DESWAP displays information about a version of a software.

With the third use case we provide the means to update the software catalog. Authorized users can edit existing entries, enter information about new versions of a software, and add software that has not been recorded so far. Since this use case enables users to edit the software catalog the user interface must reflect the representation of software in the knowledge base. For instance, the form a user to specifies the type of a software with must provide the full hierarchy of software types instead of the simplified alternative.

At the demonstration the interested audience can browse our software catalog; we will present how a user updates the catalog and guide developers to advertise their software in our system; and, most important, we will demonstrate how users specify CS requirements and how our system automatically proposes software that satisfies these requirements.

[2]http://jena.sourceforge.net
[3]http://pellet.owldl.com

[4]We currently support UML models that have been created with ArgoUML (cf. http://argouml.tigris.org).