Automatic Component Selection with Semantic Technologies

Olaf Hartig, Martin Kost, and Johann-Christoph Freytag

Humboldt-Universität zu Berlin
Department of Computer Science
(hartig|kost|freytag)@informatik.hu-berlin.de

Abstract. Selecting a suitable set of available software components for a component-based software system is a laborious task, often too complex to perform manually. We present a novel approach to automatic component selection that respects dependencies between the required components, an issue not considered by existing approaches. Our approach, which utilizes semantic technologies, is based on comprehensive semantic descriptions of software components and their functionalities.

1 Introduction

Developing software requires a systematic procedure to be successful. This holds especially for large, complex software systems. For the development of such systems the application of engineering techniques is an absolute necessity. Responding to the needs the software engineering community proposed software reuse and introduced methodologies for component-based software development (CBD). CBD is concerned with the development of software from preproduced parts, the ability to reuse those parts, and the maintainance and customization of the parts [1]. These parts, called *software component*, are units "of composition with contractually specified interfaces and explicit context dependencies only." [2]

Due to the reuse of software, CBD promises reduced development times, increased flexibility, and increased reliability of component-based systems. It is obvious, buying a component takes less time than designing, implementing, testing, debugging and documenting a component. Self-contained components that offer a defined set of functionalities to a system can be exchanged more easily. New developments are likely less mature than components that have already been used in other systems [3].

The main challenge in designing component-based systems is finding and selecting components, often denoted as the *component selection* (CS) problem [4,5]. Finding a set of candidate components for each required functionality may become a laborious task. Once a set of possible candidate components for each required functionality has been determined, a subset of all candidates must be selected that satisfies the developers' objectives. The difficulty in selecting such a subset is finding a selection where the single components are compatible with each other. Finding and selecting components will quickly become too complex to be performed manually, especially for larger systems.

To relieve designers of component-based systems of the burden of manual CS we developed an approach to solve the CS problem automatically. Our approach utilizes semantic technologies such as ontologies, rules and reasoning. To enable automatic CS we developed a comprehensive ontology that represents software components, their properties, and their functionalities as well as the CS-specific requirements. Based on this representation we developed concepts for a machine-based CS method. To evaluate our approach we implemented our concepts in a system that supports developers during the design of component-based Semantic Web applications.

This paper is structured as follows. First, in Section 2 we discuss the challenges of CBD and identify the problems that we want to solve by our machine-based CS approach. In Section 3 we introduce our ontology and in Section 4 we describe our CS method. Section 5 provides a brief description of our system that applies the presented approach. Finally, Section 6 reviews related work and Section 7 concludes this paper with a summary and an outlook to future work.

2 Challenges of Component-Based Software Development

Before we introduce our machine-based approach to CS we pinpoint the scope of the approach and we identify the main requirements. Therefore, in this section we clarify our notion of CBD and describe the main challenges thereof.

Our approach targets software systems that are implemented by the realization of various functionalities; a majority of the functionalities or variations thereof are already implemented and offered by third-party software. We propose to implement these kinds of software systems using a component-based architecture and realize as much of the functionalities by integrating existing software. Hence, our understanding of CBD considers components as architectural units [6] that offer a set of specific functionalities and that can be integrated in other systems. For instance, these units could be libraries, tools, and even Web Services.

The requirements analysis for a new system comprises the identification of functionalities that support the system. We refer to those functionalities that may be realized by exisiting components as *required functionalities*. A definition of the required functionalities becomes part of the requirements specification; we refer to this part as the *CS requirements*.

The main challenges of developing software systems using CBD principles as outlined are the following. During the design phase the developer must find and select software that satisfies the CS requirements. Furthermore, the developer must integrate the selected software in the system. Integration usually happens during the implementation phase, even if an integration strategy must be specified during design. However, since the concepts presented in this paper aim to support finding and selecting software components the remainder of this section focuses on these two tasks.

2.1 Finding Candidate Components

Given the CS requirements the developer must, for each of the required functionalities, find software that offer these functionalities. Finding candidate software is a laborious task. A large amount of software catalogs exist on the internet (e.g. SemWebCentral[1]). The majority of these catalogs offer a human-readable interface only; there scarcely are services that offer machine-processable data about the cataloged software. The user interfaces of the online catalogs vary to a great extend: the navigation structures are different, so are the search and browsing capabilities. Hence, locating candidate software is difficult. Once discovered, the developer must check if the software really offers the functionality as advertised; usually the textual descriptions do not provide the necessary level of detail. Besides the satisfaction of the functional requirements a developer may consider further properties of a software before selecting it as a candidate. For instance, non-technical requirements [7] such as the necessity for Free Software licenses or the preference of a specific producer may have an impact on the decision.

2.2 Selecting Components

The result of finding is a candidate set of software for each functionality. Using these sets, the developer must select a set of software that satisfies all of her requirements. Essentially, the set must contain at least one offering software for each functionality. Notice, a software component might offer more than one of the required functionalities.

However, selecting the set of software is not as easy as picking one software from each candidate set. Usually, the required functionalities rely on each other. For instance, a system may store the result of a remote query to a local database; since the data import format must be compatible with the format of the query result the storage functionality depends on the query functionality; selecting a query processor restricts the choices for potential data stores and vice versa. In general, the software in the selected set must be compatible with respect to the dependencies between functionalities. Apparently, these kinds of dependencies add a higher degree of complexity to the selection process. Additionally, the dependencies illustrate that specifying the selected software by a simple set is not enough; a sufficient selection must define the association of each required functionality with the offering software.

Furthermore, the decision for a satisfying selection of software might be influenced by optimality criteria. For instance, the cardinality of the selected set must be minimal [4] or the overall cost of the selected software must be minimal [5].

An additional challenge for the selection is composed software, i.e., software which is a composition of other software components. Often the functionalities of the contained components are not advertised for the composition even if they are usable in a system that integrates the composed software. For instance, the RDF framework Jena[2] contains additional Java libraries such as the XML

[1] http://www.semwebcentral.org
[2] http://jena.sourceforge.net

parser Xerces[3]; although, the feature list of the Jena package does not mention XML parsing capabilities. Obviously, finding and selecting a parser may become obsolete if the developer selects Jena for some functionality of a system that additionally requires an XML parsing functionality. Thus, to find more optimal selections the developer must consider the functionalities offered by components of composed software.

The selection of software components has become an interest of research in the software engineering community recently [5,8]. Various approaches build on existing research of CS problems in other engineering disciplines such as industrial design [9]. However, to the best of our knowledge none of the presented approaches considers the compatibility requirements between the selected software components (i.e. the dependencies described here).

3 Representing Software and Requirements

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

3.1 Software

Figure 1 illustrates some of the main concepts in our ontology that deal with software. We classify *software* in a hierarchy of *software types*. For instance, DB2[4] is a relational database management system (RDBMS); RDBMSs are a special kind of database managements systems (DBMS). We distinguish different versions of a software by the concept of a *software release*. Each software release offers certain *functionalities*. These functionalities are classified in a second hierarchy, the hierarchy of *functionality types* (e.g. importing data is a special kind of adding data). 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 release has specific values for its functionality properties. While, for instance, the supported import format is a property of data import functionalities in general, version 9.5 of DB2 offers a particular data import functionality which supports the IXF format [10] as its import format.

Besides the aforementioned concepts for software we model composed software, dependencies of software, and many other properties such as licenses and prices. Additionally, we introduce *composition constraints* that specify conditions under which functionalities offered by software releases can be combined

[3] http://xerces.apache.org
[4] http://www.ibm.com/db2

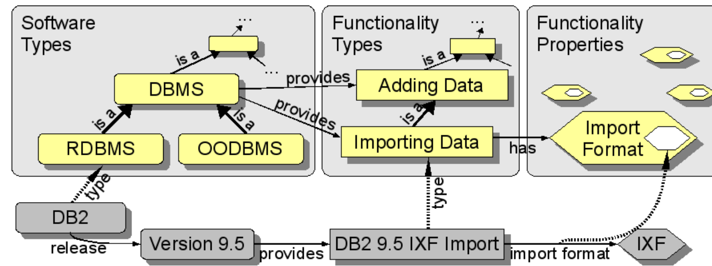

Fig. 1. Extract of a software catalog.

without conflict. For each ordered pair of functionality types a composition constraint identifies those functionality properties that must have mutually compatible values. For instance, a composition constraint for storage functionalities that depend on query functionalities specifies that the storage import format must be compatible with the query result format. We use composition constraints to verify whether a selection of software releases is compatible with respect to the dependencies between required functionalities.

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 CS approach utilizes these meanings to discover potential software components (cf. Section 4.2).

3.2 Requirements

In addition to software, our ontology represents *CS requirements* which contain required functionalities and dependencies between the required functionalities. *Required functionalities* are those functionalities of a component-based system that may be realized by components. We specify a required functionality 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 release that offers functionalities of this type could potentially be selected as the 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 releases that offer a functionality with the permitted properties may realize the required functionality.

Required functionalities may depend on each other as discussed before (cf. Section 2.2). We represent *dependencies* between required functionalities as a part of the CS requirements; each dependency is a pair of required functionalities where the second required functionality depends on the first one.

4 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 release that offers a functionality which can implement the required functionality. A selection of software releases that satisfies all CS requirements is a *global solution*. To satisfy all CS requirements a selection must associate every required functionality with a software release 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 releases must not violate the composition constraints for the respective dependencies. Hence, our CS problem is the following: given CS requirements and a software catalog, find a global solution for the requirements with software releases from the catalog.

A naive approach to find a global solution is to iterate over all selections that combine exactly one software release from each local solution until a selection is found that does not violate the dependencies. In the worst case, this method generates the whole search space which is too inefficent. Especially for complex requirements with many dependencies only very few of the possible combinations qualify as satisfying selections.

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 constraint 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 constraints 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 release from a set of local solutions for the global solution prescribes an implementing functioniality for the respective required functionality. Hence, selecting a local solution for a required functionality yields additional restrictions for the required functionality. By propagating these restrictions we can reduce the local solutions even further.

Based on our propagation approach we propose a method that consists of three main steps (cf. Figure 2): the propagation of restrictions, the identification of local solutions, and the identification of a global solution. In the following we describe these steps in detail and review our approach in the context of constraint satisfaction problems.

4.1 Propagate Restrictions

Figure 3 illustrates an algorithm that utilizes composition constraints to propagate property restrictions. Since propagation is based on composition constraints it can only be applied to required functionalities that are in a dependency relationship. The algorithm expects a set of required functionalities RF, a set of

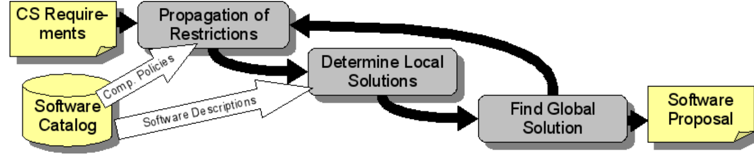

Fig. 2. The main steps of our CS method.

dependencies D, and a set of composition constraints CC. By $type(rf)$ we denote the functionality type for a required functionality $rf \in RF$; for each property restriction pr of a required functionality, $fctProp(pr)$ denotes the restricted functionality property and $vals(pr)$ denote the values permitted for $fctProp(pr)$. The set of dependencies $D \subseteq RF \times RF$ contains a pair (rf_a, rf_b) for each dependency relationship where $rf_b \in RF$ depends on $rf_a \in RF$.

The algorithm repeatedly iterates over all dependencies as long as it is possible to propagate property restrictions. The algorithm terminates because the

Algorithm: Propagate
Input: RF – required functionalities; D – dependencies; CC – composition constraints
Output: `TRUE` – propagation successful, `FALSE` – propagation failed

```
Tag all property restrictions of all rf ∈ RF as new;
LET propagated := TRUE;
WHILE propagated = TRUE DO
    LET propagated := FALSE;
    FOREACH (rf_a, rf_b) ∈ D DO
        LET cc the composition constraint for (type(rf_a), type(rf_b));
        LET NPR the set of all new property restrictions of rf_a;
        FOREACH npr ∈ NPR DO
            FOREACH fp_b ∈ {fp | cc has a condition for (fctProp(npr), fp)} DO
                IF rf_b has property restrictions for fp_b THEN
                    LET opr_b the property restrictions of rf_b for fp_b;
                    IF vals(opr_b) incompatible to vals(npr) THEN
                        RETURN FALSE;
                    ELSE IF npr is more restrictive than opr_b THEN
                        Remove opr_b from the property restrictions of fp_b;
                        LET npr_b := (fp_b, vals(npr));
                        Tag npr_b as new;
                        Add npr_b to the property restrictions of rf_b;
                        LET propagated := TRUE;
                ELSE
                    LET npr_b := (fp_b, vals(npr));
                    Tag npr_b as new;
                    Add npr_b to the property restrictions of rf_b;
                    LET propagated := TRUE;
RETURN TRUE;
```

Fig. 3. Algorithm that propagates property restrictions.

set of dependencies is finite, so are the sets of property restrictions; each propagation replaces a less restrictive restriction by a more restrictive one and, hence, reduces the respective number of permitted values. Notice, propagation may fail if the permitted values in a propagated restriction are incompatible with the values currently permitted.

4.2 Determine Local Solutions

To determine the set of local solutions for a required functionality we propose to query the software catalog with a query generated from the required functionality. In the DESWAP system (cf. Section 5) we generate SPARQL queries because the software catalog is realized as an RDF repository with OWL descriptions. Consider, for instance, a required data import functionality that has a property restriction which permits IXF as supported import format. For such a required functionality we basically generate a query that is similar to the query in Figure 4. However, the actual queries generated by our system are more complex for the following reason. As discussed in Section 2.2 composed software releases may contain third-party software components that offer functionalities not advertised for the releases themself. Our ontology enables the description of these cases. Accordingly, our system generates queries that additionally consider composed software releases.

```
SELECT ?sr
WHERE {
   ?sr deswap:release_has_ServiceAssociation ?sa .
   ?sa deswap:serviceAssociation_has_Functionality ?fct .
   ?fct rdf:type deswapType:ImportingData ;
        deswap:functionality_supports_import_format ?ifmt .
   FILTER ( ?ifmt == deswapParam:XFI )
}
```

Fig. 4. SPARQL query that determines local solutions (prefix definitions omitted).

It is not always obvious that a software release offers a functionality which can implement a required functionality. For instance, the software catalog may contain the information that version 9.5 of DB2 offers an importing data functionality (cf. Figure 1). Additionally, the catalog may contain the information that importing data functionalities are a special kind of adding data functionalities. From these two facts we can infer that DB2 9.5 offers an adding data functionality. Hence, DB2 9.5 may realize a required adding data functionality even if this is not stated explicitly in the software catalog. We can discover such implicit knowledge automatically. Since the software catalog is based on our ontology the descriptions in the catalog have a machine-processable meaning. A reasoner uses these semantic descriptions to discover implicit facts about the software, its functionalities, and the supported functionality properties. These additional facts enable more complete sets of local solutions.

```
Algorithm: DetermineGlobalSolution
Input: RF – required functionalities; D – dependencies; SC – software catalog;
       CC – composition constraints
Output: the global solution or nothing

LET success := Propagate( RF, D, CC ); // propagate user-specified restrictions
IF success = TRUE THEN
    Determine the set of local solutions for all rf ∈ RF;
    LET TMP := {};
    LET success := CompleteGlobalSolution( RF, D, SC, CC, TMP, 1 );
    IF success = TRUE THEN
        RETURN TMP;
RETURN;
```

```
Algorithm: CompleteGlobalSolution
Input: RF – required functionalities; D – dependencies; SC – software catalog;
       CC – composition constraints; TMP – partial global solution; i – recursion depth
Output: TRUE – global solution completed, FALSE – impossible to complete global solution

Backup property restrictions of all rf ∈ RF;
Backup the set of local solutions for all rf ∈ RF;
LET rf_i the ith element of RF;
LET LS the set of local solutions for rf_i;
FOREACH ls ∈ LS DO
    LET f the functionality of ls that can implement rf_i;
    Replace the property restrictions of rf_i by new property restrictions generated from f;
    LET success := Propagate( RF, D, CC );
    IF success = FALSE THEN
        Restore property restrictions for all rf ∈ RF;
    ELSE
        Update local solutions for all rf ∈ RF;
        IF at least one set of local solutions is empty THEN
            Restore the set of local solutions for all rf ∈ RF;
            Restore property restrictions for all rf ∈ RF;
        ELSE
            Add (rf_i, ls) to TMP;
            IF i = |RF| THEN
                RETURN TRUE;
            ELSE
                LET success := CompleteGlobalSolution( RF, D, SC, CC, TMP, i + 1 );
                IF success = TRUE THEN
                    RETURN TRUE;
                ELSE
                    Remove (rf_i, ls) to TMP;
                    Restore the set of local solutions for all rf ∈ RF;
                    Restore property restrictions for all rf ∈ RF;
RETURN FALSE;
```

Fig. 5. Recursive algorithm that determines a global solution.

4.3 Find Global Solution

To determine a global solution for our CS problem we propose the algorithm illustrated in Figure 5. The algorithm recursively 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. If a set of local solutions becomes empty during the iteration it is impossible to construct a global solution with the candidates that have already been selected. 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. In the latter case it is impossible to find a selection that satisfies all CS requirements because either the software catalog does not contain enough software releases or the user-specified property restrictions are too strict.

4.4 Component Selection as a Constraint Satisfaction Problem

It is possible to view our CS problem as a constraint satisfaction problem. Constraint satisfaction problems are defined by a set of variables, a set of possible values for each variable, and a set of constraints restricting the values that the variables can simultaneously take; the solution to a constraint satisfaction problem is a mapping that assigns every variable one of its possible values and that does not violate the constraints [11]. In our case the variables are the required functionalities, the possible values are the local solutions, and the constraints are the composition constraints that must hold for dependent required functionalities. Constraint programming deals with solving constraint satisfaction problems. Barták [12] classifies constraint programming techniques. In Barták's terminology our CS approach is a combination of systematic search and a consistency technique that removes inconsistent values until a solution is found; to remove inconsistent values we apply a full look ahead technique that propagates constraints.

5 The DESWAP System

We implemented our concepts in the DESWAP[5] system which is primarily intended to be used for component-based systems that apply Semantic Web technologies. The main features of the DESWAP system are:

- a *machine-accessible software catalog* with OWL descriptions of software components that could be integrated in Semantic Web applications,
- a *Web-based user interface* to the software catalog that hides the complexity of software descriptions, and
- a *sophisticated CS tool* that enables the specification of CS requirements and that proposes a suitable selection of software components which satisfies the specified requirements.

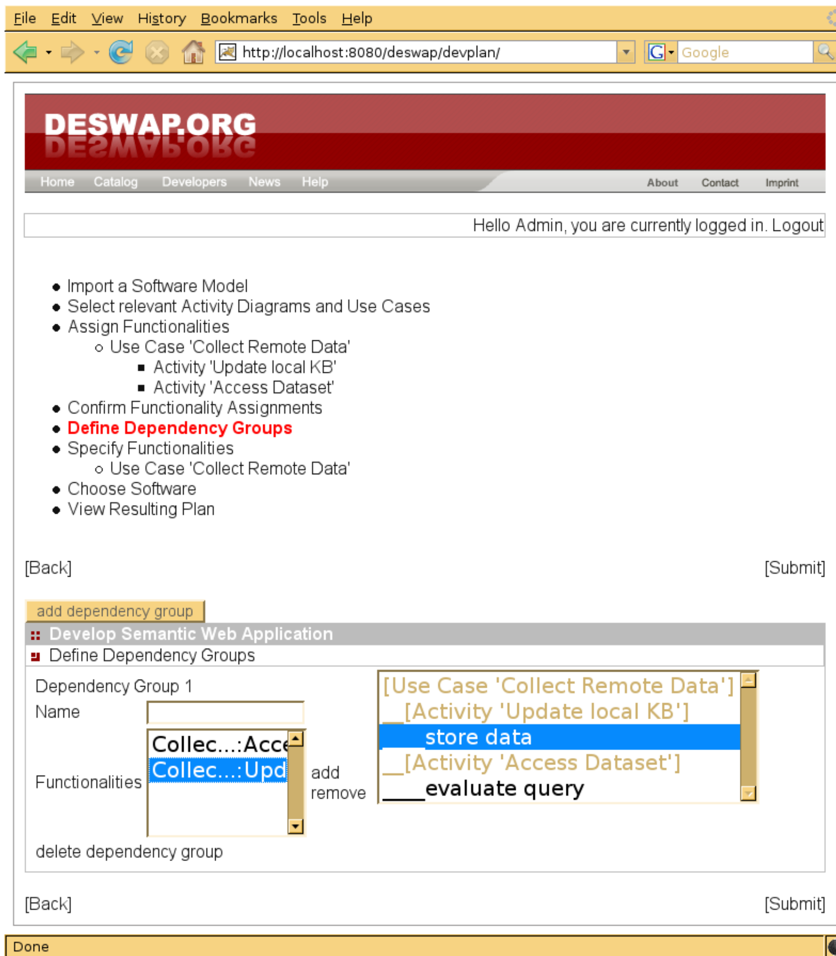

Fig. 6. Defining dependencies between required functionalities with DESWAP.

We implemented the DESWAP system as a JSP-based Web application that accesses the DESWAP knowledge base with the Jena framework. The knowledge base itself consists of five OWL documents, an RDF repository, the domain-specific composition constraints, and a reasoner; the OWL documents define our ontology, the RDF store contains OWL descriptions of the software catalog, and the reasoner, Pellet[6] in our case, discovers implicit knowledge. A SPARQL endpoint provides machine-based access to the knowledge base. For human users we provide a Web-based interface for browsing as well as editing the data about software releases in the catalog. In the future, we will provide a user interface

[5] Development Environment for Semantic Web APplications

[6] http://pellet.owldl.com

to enable a limited group of authorized users to update the software types, the functionality types, and the constraints.

The CS tool of DESWAP 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 design the software before implementing it; they create a software model which defines the use cases and the activities that realize each use case. Based on our understanding of a component-based system (cf. Section 2) 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. DESWAP 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 6).

Developers will use their software model to specify the CS requirements. In order to integrate our system in the development process we support the import of software models[7]. After supporting the users to specify their CS requirements DESWAP applies our CS method and determines a suitable selection of software components which satisfies the requirements.

6 Related Work

CS problems are investigated in various engineering disciplines. Fox et al. [4] define the component selection problem as "the problem of choosing the minimum number of components from a set of components such that their composition satisfies a set of objectives."

The only approach, to the best of our knowledge, that considers compatibility requirements between selected components has been proposed in the context of industrial design. Carlson [9] defines a component selection problem for the design of engineering systems. The designed systems consist of a set of generic components that must be implemented by exisiting components. From a manufacturers' catalog an engineer chooses a set of components for the implementation. Choosing a specific component for one task may have an effect on the other components. This kind of dependency is similar to the dependencies in our case where the selection of a specific software release for a required functionality could possibly add further restrictions on other required functionalities. Carlson proposes the application of genetic algorithms to solve her problem. To evaluate the possible solutions a simulation of the system is performed. Hence, the possible dependencies between components are not considered explicitly. Our approach, in contrast, considers the dependencies during the construction of solutions.

We focus on the selection of software components. CS problems have become an interest of research in the software engineering community recently. For instance, Haghpanah et al. [5] consider the cost of software components. For a set of requirements they try to find a satisfying set of components with minimal

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

overall cost. Since the problem is NP-complete the authors propose and evaluate a greedy algorithm and a genetic algorithm that approximate an optimal solution. However, Haghpanah et al. do not consider dependencies between the requirements.

An approach to support component-based development with semantic technologies similar to our DESWAP system has been presented by Inostroza and Astudillo [8]. The authors outline a conceptual framework to characterize software components with respect to their non-functional properties. The framework enables the selection of suitable software components for non-functional requirements. Even if Inostroza and Astudillo propose ontologies to describe software components as we do the characterization is limited to non-functional properties. However, as in the DESWAP system, the authors distinguish two groups of users that provide different kinds of information. A limited community of experts provides controlled descriptions for non-functional properties. A wider distributed community describes software components using the existing non-functional property descriptions. In their paper Inostroza and Astudillo focus on the concepts of the proposed ontology and the relationships of these concepts; what is missing is a detailed discussion of a method to select suitable components for non-functional requirements.

7 Conclusion

Existing approaches for the automatic selection of software components do not consider compatibility requirements between the selected components. These requirements add a high degree of complexity to the selection process. In this paper we present a novel approach that respects this kind of dependencies. We propose an algorithm that finds a selection which satisfies the specified requirements. Our approach uses semantic technologies to represent available software components and to solve the component selection problem.

We currently do not consider optimality criteria such as a minimal number of software in the solution. However, we are working on an extension of our algorithm to find optimal selections. To develop concepts for a suitable extension we are studying methods that solve constraint satisfaction optimization problems [12].

References

1. Heineman, G.T., Councill, W.T., eds.: Component-Based Software Engineering: Putting the Pieces Together. Addison-Wesley Longman Publishing Co., Inc. (2001)
2. Szyperski, C.: Component Software: Beyond Object-Oriented Programming. 2nd edn. Addison-Wesley Longman Publishing Co., Inc. (2002)
3. Clements, P.C.: From Subroutines to Subsystems: Component-Based Software Development. The American Programmer **11**(8) (1995)
4. Fox, M.R., Brogan, D.C., Reynolds, P.F.: Approximating Component Selection. In: ACM/IEEE Winter Simulation Conference. (2004) 429–435

5. Haghpanah, N., Moaven, S., Habibi, J., Kargar, M., Yeganeh, S.H.: Approximation Algorithms for Software Component Selection Problem. In: Proceedings of the 14th Asia-Pacific Software Engineering Conference (APSEC). (2007) 159–166
6. Lau, K.K., Wang, Z.: Software Component Models. IEEE Transactions on Software Engineering **33**(10) (2007) 709–724
7. Carvallo, J.P., Franch, X.: Extending the ISO/IEC 9126-1 Quality Model with Non-Technical Factors for COTS Components Selection. In: Proceedings of the International Workshop on Software Quality (WoSQ). (2006) 9–14
8. Inostroza, P., Astudillo, H.: Emergent Architectural Component Characterization using Semantic Web Technologies. In: Proceedings of the 2nd International Workshop on Semantic Web Enabled Software Engineering (SWESE). (2006)
9. Carlson, S.E.: Genetic Algorithm Attributes for Component Selection. Research in Engineering Design **8**(1) (1996) 33–51
10. IBM: Data Movement Utilities Guide and Reference. DB2 Version 9.5 Manuals. (2008)
11. Tsang, E.: Foundations of Constraint Satisfaction. Academic Press, London and San Diego (1993)
12. Barták, R.: Constraint programming: In pursuit of the holy grail. In: Proceedings of the Week of Doctoral Students (WDS). (1999) 555–564