

An Extensible Language for Service Dependency Management

Siamak Haschemi and Arif Wider

Institut für Informatik

Humboldt-Universität zu Berlin, Unter den Linden 6

10099 Berlin, Germany

{haschemi|wider}@informatik.hu-berlin.de

Abstract—Service dependency management in service-oriented component platforms is described with languages, which cannot be easily adapted to domain-specific requirements. This prevents the development of language concepts, which are more suitable for the cognitive space and intuition of domain experts than general-purpose languages. We use a model-driven approach to create an extensible language supporting the creation of new language concepts. With this approach, languages for service dependency management can be extended to have the desired concepts and expressiveness.

Keywords—metamodeling; domain-specific languages; service dependency management;

I. INTRODUCTION

Automatic software composition is needed by complex systems created with component-based approaches. This is especially important for dynamic software applications in which software functionality is added and removed without the control of the application. These applications need to be able to deal with dynamically available software.

Over the past years, service-oriented concepts have become popular for designing applications. In applying these concepts to concrete technologies, service-orientation and component-orientation have been combined in platforms called *service-oriented component platforms*. Within these platforms services are provided by components and can be used by other components. Services can be published and removed at any time and this is not under the control of the service user. Service-oriented applications must therefore deal with dynamic availability of services. In *automatic service dependency management* (ASDM), components are attached with metadata describing their service dependencies. At runtime, this description is used to control the life cycle of the component: It is started only if all dependencies can be resolved, and stopped if one of them cannot be resolved. The resolvability of dependencies is checked on arrival or departure of provided services. One of the advantages of this approach is that the component source code is freed from dependency management aspects, which makes the development of dynamic applications manageable.

Software systems are used in many different domains. In each of these domains, specific and sometimes unique concepts and languages emerged over time. In many cases, domain experts are not familiar with general-purpose programming or modeling languages like Java or UML. Therefore, the help of software engineers is needed to

create new systems. *Domain-specific languages* (DSL) try to fill this gap between the concepts known by the domain experts, and the concepts of the target system. They contain only concepts necessary for a certain domain.

In this paper, we explore the domain-specific extensibility of description languages for ASDM. We create a DSL, which contains common concepts for describing service dependencies. In contrast to existing approaches, this language can be extended with domain-specific concepts.

The next section provides an overview of model-driven development (sec. II). We identify common concepts for service dependency management and create an extensible description language in section III. Section IV demonstrates the extensibility of the created language. An overview of related work is presented in section V, while section VI presents future work and concludes this paper.

II. MODEL-DRIVEN DEVELOPMENT

Model-driven development (MDD) is a methodology in software engineering that focuses on the creation and the processing of *models*. Models are the primary artifacts in MDD. They are not just used for documentation but play the role of source-code [1]. Models aim at a high level of abstraction. Therefore, models often have a rather declarative and a less imperative character. Models are expressed in modeling languages. There are general-purpose modeling languages (e.g., UML) and *domain-specific languages* (DSLs). A DSL is a language that is specially tailored to the needs of a certain problem domain. The goal of a DSL is to be particularly expressive in its domain, i.e., common concepts of the domain can be expressed concisely.

When developing a DSL, one has to specify its *abstract syntax*, its *concrete syntax* and its *semantics*. The domain's concepts and their relations are defined by the DSL's abstract syntax. The abstract syntax can be described by a *metamodel*. Models, that are expressed in a DSL are instances of the DSL's metamodel. In order to allow the creation of models, a DSL has to provide a concrete syntax. The concrete syntax defines how the concepts of the domain are represented, e.g., by certain keywords. A DSL can provide more than one concrete syntax, e.g., a graphical and a textual syntax. A common way of describing the semantics of a DSL is to define transformations to a target language. A target language can be an executable language like Java.

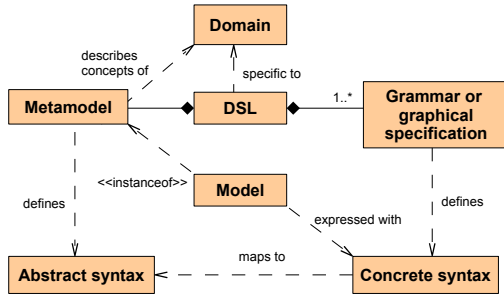


Figure 1. MDD terminology

In this paper, we show the metamodel-based development of a DSL. Therefore, we create a metamodel and define a concrete textual syntax by providing a grammar definition which maps to the metamodel elements. This metamodel-based approach lets us leverage the advantages of existing metamodel-based technologies [2], [3]. With these technologies it is possible to generate editing or debugging tools from the metamodel and the grammar definitions of a DSL. Today, such tooling is crucial for the success and the usability of a language. Because a DSL is used by much less users than a general purpose language, the effort for providing sufficient tooling has to be small. As an example, we demonstrate the automatic generation of an editor that provides syntax highlighting, code completion and syntax related checks.

III. CONSTRUCTING THE LANGUAGE

In this section, we demonstrate the construction of a DSL for the domain of service dependency management. First, we extract common concepts of this domain (sec. III-A). Second, we create a metamodel that describes these concepts (sec. III-B). Third, we define a concrete textual syntax by providing a grammar (sec. III-C).

A. Identifying Concepts

In service-oriented component platforms, dependencies between components are described at a level of services. Hence, components provide and require services. Furthermore a component can require more than one service of a kind. Using ASDM, a component is the unit of resolvability, i.e., depending on availability of required services a component is either stopped or started. If a component is stopped, its provided services are not available.

Services are described by a *service specification* and service metadata. The service specification is an interface type that provides a set of operations. The interface describes how a service is used and this is not specific to a concrete implementation of that interface. This is how loose coupling between service providers and service consumers is achieved.

Service metadata describes a concrete service implementation, i.e., it provides Quality of Service (QoS) information about that service implementation. This metadata is often expressed as key-value pairs and is referred to as *service properties* [4]. Most ASDM frameworks separate the description of service dependencies and service metadata

from the service specification¹. Therefore a component description usually references service specifications that are expressed in an interface definition language (IDL).

A required and a provided service only match, if they reference the same service specification. With the help of service properties, further quality requirements can be expressed. This is usually done by using some kind of *filter* language. If more provided services meet the requirements than needed, it has to be decided which services are selected. Therefore, some ASDM frameworks introduce *service rankings* to define criteria that determine the selection of services in such a case. Bottaro and Hall showed, that definitions for service rankings can get complex and that it is domain-specific what means are needed for an adequate service ranking description [5].

Summarizing, the following concepts can be identified:

- Component: subject of service dependencies, unit of resolvability.
- One component can provide and require services.
- Separation of service specification and service dependency description.
- Service properties describe non-functional characteristics of a concrete service implementation (QoS).
- Filters using service properties allow expressing complex service requirements.
- Service rankings determine an order for available services that meet the requirements.

B. Metamodeling the Domain

In section III-A, we identified common concepts for automatic service dependency management. We model these concepts in a metamodel through UML class diagrams. To use advanced tools from the Eclipse MDD community, we describe a model-transformation from this class diagram to an EMF² Ecore Metamodel (We will not show the transformation details in this paper). Fig. 2 shows an overview of the developed metamodel.

- A *Component* describes service dependencies and provided services.
- A *ServiceProvision* is used to provide a service. It contains service properties, identified by their names (*name*) and types (*value*). We refer to the service type in the attribute *specification*. We therefore imply that the service type is defined in the target platform (e.g., in the OSGi Service Platform [6] the service type is described with Java interfaces).
- A *ServiceRequirement* represents a service dependency. It consists of a binding cardinality (*Multiplicity*), a filter expression (*Filter*), and a ranking method (*ServiceRanking*).
- A *Multiplicity* represents a limitation for bounded service providers. It defines an interval between *lower* and *upper*, with *lower* being either *Optional* or *Mandatory*, and *upper* being defined by an integer (*BoundedUpper*) or being unbounded (*UnboundedUpper*).

¹e.g., the Newton Framework (<http://newton.codecauldron.org>)

²Eclipse Modeling Framework (<http://www.eclipse.org/modeling/emf/>)

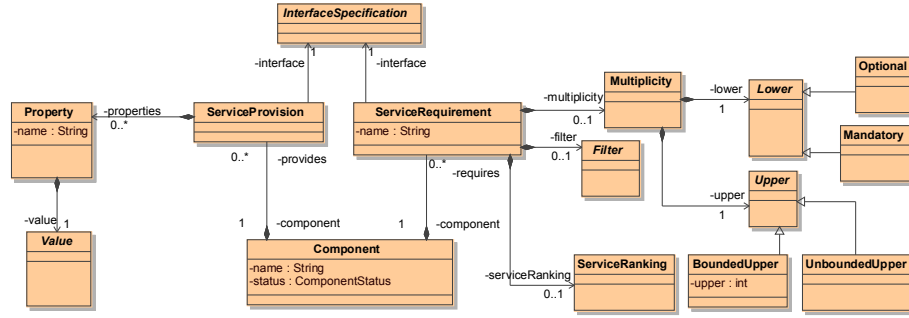


Figure 2. The DSL metamodel (Metamodel of the extensible language).

- A *Filter* narrows the set of available services according to the service requester needs [5]. We create a metamodel containing the language concepts of LDAP filters (which are also used in OSGi) to express filtering. We will skip the details and refer to the RFC³.
- The *ServiceRanking* is used to rank all filtered services in a total order according to the adequacy of their ability to meet the service requirements [5]. We will go into details of service ranking methods in section IV.

C. Defining a Concrete Textual Syntax

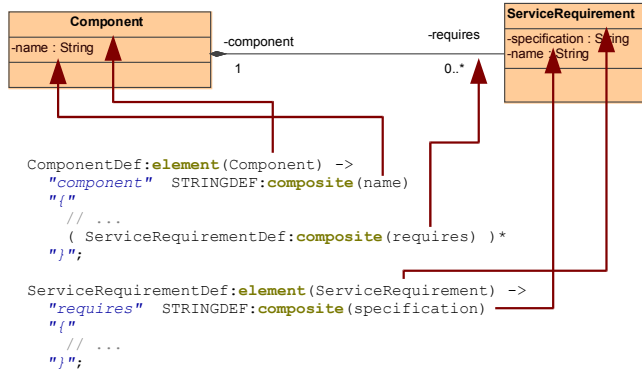


Figure 3. TSL definition with related metamodel elements.

In the previous section we show the structure of the developed metamodel. To create model instances of our metamodel, we need to create a concrete syntax. We choose the *Textual Editing Framework* (TEF) [2] to develop a textual syntax for the description of service dependencies. TEF utilizes generative engineering techniques to make text editor development for small applications practical. In TEF, a language engineer describes the textual syntax in a language called TSL. TEF uses a TSL definition and creates an enhanced Eclipse editor with features like syntax highlighting, outline views, annotation of syntactical and semantic errors, occurrence markings, content-assist, and code formatting. TSL is based on (E)BNF-grammars. The rules and symbols in a TSL description refer to according metamodel elements. In Fig. 3 parts of our TSL definition and the related metamodel elements are

shown. The typical BNF-grammar (with terminal symbols and nonterminals) is extended with references to classes, references, and attributes. Based on this grammar, TEF parses the editor input and creates model instances consisting of metamodel elements.

D. Extensibility of the DSL

Our DSL has been created with the help of a metamodel and an extended BNF grammar to describe the concrete textual syntax. A domain-specific extension of the DSL could therefore be created by providing new versions of these two artifacts. The grammar would typically need new rules, which can be added to the TSL definition. The new metamodel would extend the presented metamodel and add its domain-specific concepts. There exists various techniques to extend metamodels, including approaches from the UML (a.k.a. package merge, subsets, redefinition) and aspect-oriented approaches [7]. We can use these techniques to apply domain-specific language concepts, and create enhanced textual editors for the new DSL with short delays.

IV. AN EXEMPLARY EXTENSION OF OUR LANGUAGE

In section III-A we identified service rankings as a common concept in ASDM. Because the definition of service rankings often is a domain-specific task, we decided not to integrate concrete means for service ranking description into our language in the first place. Here, we show one approach that uses boolean-valued LDAP-like expression which are extended with a means for comparing properties of two services.

Therefore we extend the metamodel with a *ServiceRankingExpression* that derives from *ServiceRanking*. This expression consists of a filter. The difference of this filter to the existing LDAP-like filter is the availability of the new operation *AttributeComparison*. This operation allows distinguishing between properties of two services. The possibility to compare services allows ordering a set of services. This resembles the comparator design pattern. Along with the metamodel, we modified the grammar to generate a new Eclipse editor (Fig. 4).

There are advantages of this approach over the common approach of referencing a comparator-like method that contains arbitrary code to determine a service ranking. The complexity of arbitrary Java code for instance is nearly unlimited. That makes reasoning about service

³<http://www.ietf.org/rfc/rfc1960.txt>

ranking descriptions of that kind not feasible. In contrast, expressions in a small and well-defined language allow static analysis, which can be used to support a resolving mechanism with useful hints.

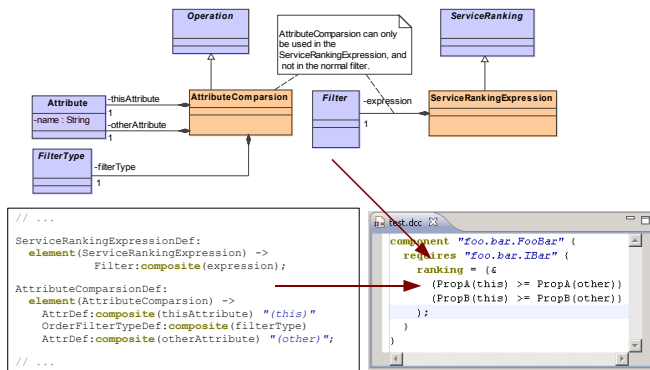


Figure 4. Supporting service ranking through filter expressions.

V. RELATED WORK

Several approaches for describing automatic software composition exists in the literature [5], [8]–[10]. In these languages, a system is described in terms of loosely coupled components with provided and required services.

A language for contextual service composition is proposed in [9]. It contains a XML-based language to describe dynamic service composition with service selection based on attribute-value pairs, including filters, and utility functions. While dynamic service availability and service selection is addressed by this work, it does not support a concrete textual syntax other than XML.

Several frameworks for automatic service dependency management exists for the OSGi Service Platform [6]. Pioneer work in this area has been done by Cervantes and Hall with the creation of the Service Binder framework [11]. The ideas presented in this work led to the OSGi Declarative Service Specification [10]. A new momentum has been introduced by Escoffier and Hall with the iPOJO framework [8]. All of these frameworks use similar concepts to integrate ASDM into OSGi. They use a XML-based description language to describe service dependencies. Service Binder and OSGi Declarative Services lack extensibility of their description languages. However, iPOJO is somehow extensible through handlers. But compared to our work, the domain-specific extension of the iPOJO language is complex and highly tied to the framework. Our model-based approach for domain-specific extensions, in contrast, is independent from the concrete syntax and target platform. We think that we can use these frameworks as target platforms for our language.

VI. CONCLUSION AND FUTURE WORK

In this paper, we applied a metamodel-based approach to construct an extensible language for service dependency management. We created a metamodel for the domain of service dependency management in service-oriented component platforms. Therefore, we identified concepts that are commonly used in this domain. By defining a grammar that references elements of the metamodel, we

provided an exemplary concrete textual syntax for our language. A rich-featured editor for our language could be generated from the grammar definition and the metamodel. Furthermore we demonstrated that the metamodel-based approach of constructing the language allows easy extensibility of that language. Therefore, we provided an exemplary extension to the language.

Currently we are working on possibilities of using models of our language at runtime to support automatic service dependency management. Therefore, we are using the iPOJO framework [8] as a target platform for our language. By integrating our language with that framework, we want to demonstrate that the information captured by our models can be used to improve runtime service matching. For the future, we plan to investigate further possibilities of reasoning about the models of our language.

ACKNOWLEDGMENT

We like to thank Dirk Fahland, Daniel Sadilek, Guido Wachsmuth, Stephan Weißleder, and the anonymous reviewers for comments on a preliminary version of this paper. This work is supported by grants from the Deutsche Forschungsgemeinschaft, Graduiertenkolleg METRIK (GRK 1324).

REFERENCES

- [1] T. Stahl, M. Voelter, and K. Czarnecki, *Model-Driven Software Development: Technology, Engineering, Management*. Wiley & Sons, 2006.
- [2] M. Scheidgen, “Textual modelling embedded into graphical modelling.” Springer, Berlin, 2008, vol. 5095, pp. 153–168.
- [3] D. Sadilek and G. Wachsmuth, “Prototyping visual interpreters and debuggers for domain-specific modelling languages,” in *European Conference on Model-Driven Architecture-Foundations and Applications, Ecmda-Fa 2008, Berlin, Germany, Proceedings*. Springer, 2008.
- [4] P. Lalanda and C. Marin, “A domain-configurable development environment for service-oriented applications,” *IEEE Software*, vol. 24, no. 6, pp. 31–38, 2007.
- [5] A. Bottaro and R. S. Hall, “Dynamic contextual service ranking,” in *Software Composition, 6th International Symposium, SC 2007*, ser. Lecture Notes in Computer Science, vol. 4829. Braga, Portugal: Springer, 2007, pp. 129–143.
- [6] *OSGi Service Platform Core Specification, Release 4, Version 4.1*, OSGi Alliance, April 2007.
- [7] A. M. Reina Quintero and J. Torres Valderrama, “Using aspect-orientation techniques to improve reuse of metamodels,” *Electron. Notes Theor. Comput. Sci.*, vol. 163, no. 2, pp. 29–43, 2007.
- [8] C. Escoffier and R. S. Hall, “Dynamically adaptable applications with iPOJO service components,” in *6th International Symposium on Software Composition (SC 2007)*, Braga, Portugal, March 2007.
- [9] C. Funk, J. Mitic, and C. Kuhmuench, “DSCL: A language to support dynamic service composition,” in *1st IEEE International Workshop on Services Integration in Pervasive Environments*, Lyon, France, 2006.
- [10] *OSGi Declarative Services Specification, Service Platform Service Compendium, Release 4, Version 4.1*, OSGi Alliance, 2007.
- [11] H. Cervantes and R. S. Hall, “Automating service dependency management in a service-oriented component model,” in *6th Workshop on Component-Based Software Engineering*, September 2003, pp. 379–382.