I3-D5

Prototype component models and composition technology toolset for integration of logic-programming-like REWERSE languages

Project title:	Reasoning on the Web with Rules and Semantics
Project acronym:	REWERSE
Project number:	IST-2004-506779
Project instrument:	EU FP6 Network of Excellence (NoE)
Project thematic priority:	Priority 2: Information Society Technologies (IST)
Document type:	D (deliverable)
Nature of document:	P (prototype)
Dissemination level:	PP (restricted to FP6 participants)
Document number:	IST506779/Dresden/I3-D5/D/PP/a1
Responsible editors:	Jakob Henriksson
Reviewers:	Gerd Wagner
Contributing participants:	Dresden, Malta, Nancy
Contributing workpackages:	I3
Contractual date of deliverable:	31 August 2005
Actual submission date:	20 September 2005

Abstract

Any software composition system requires, among other things, a component model, which describes how components look and how they can be interfaced with each other. Previously, the component models for languages have been hand-written. We demonstrate a prototype tool for automatic derivation of a component model for any language given the description of the language as a meta-model in the Web Ontology Language OWL. The Semantic Web endeavor has given rise to new declarative languages which will require composition and reuse in the same way as in the traditional software engineering community. Thus we focus on deriving component models for declarative languages that are of importance for the Semantic Web such as OWL and the XML query language Xcerpt. We demonstrate how component models are generated for these languages using our prototype.

Keyword List

invasive software composition, component-based systems, component adaptation, metamodeling, semantic web

Project co-funded by the European Commission and the Swiss Federal Office for Education and Science within the Sixth Framework Programme.

© *REWERSE 2005.*

Prototype component models and composition technology toolset for integration of logic-programming-like REWERSE languages

Uwe Assmann[1], **Jakob Henriksson**[1] **and Ilie Savga**[1]

[1] Faculty of Informatics, Technical University of Dresden
Email: {uwe.assmann|jakob.henriksson|ilie.savga}@inf.tu-dresden.de

20 September 2005

Abstract
Any software composition system requires, among other things, a component model, which describes how components look and how they can be interfaced with each other. Previously, the component models for languages have been hand-written. We demonstrate a prototype tool for automatic derivation of a component model for any language given the description of the language as a meta-model in the Web Ontology Language OWL. The Semantic Web endeavor has given rise to new declarative languages which will require composition and reuse in the same way as in the traditional software engineering community. Thus we focus on deriving component models for declarative languages that are of importance for the Semantic Web such as OWL and the XML query language Xcerpt. We demonstrate how component models are generated for these languages using our prototype.

Keyword List
invasive software composition, component-based systems, component adaptation, metamodeling, semantic web

Contents

1 Introduction **1**

2 Background **2**
- 2.1 Language modeling levels . . . 2
- 2.2 Invasive software composition . . . 3
- 2.3 Reuse languages and component models . . . 4

3 Deriving a component model **5**

4 Prototype: HoMoGen **6**
- 4.1 Upper ontologies for language constructs and component models . . . 7
- 4.2 HoMoGen . . . 8
  - 4.2.1 Deriving generic concepts . . . 11

5 Examples **12**
- 5.1 Component model for Xcerpt . . . 12
- 5.2 Component model for OWL . . . 16

6 Conclusions **19**

7 Future work **19**
- 7.1 Problems . . . 20

1 Introduction

Developing software from smaller existing parts is called *component based development* [17] and has been around for a long time in the software engineering community [14]. There are many benefits to be harvested from creating software based on components and this method is considered a vital part of large mature systems [15]. Among other things, creating software from components allow for reuse of code. A component is a piece of software written in a language, keeping in mind that it will be used as a part of a more complete system. It is constructed in such a way that it allows other software components to connect to it through declared interfaces, i.e., to be composable.

In order to allow for the composition of software components, to have a *composition system* (Figure 1), three different things need to be specified. What we need is a *component model*, a *composition technique* and a *composition language*. A component model defines how the software components should look in order to be usable in the system. Specifically, the component model should explicitly describe the interfaces of the components, e.g., how different components can be connected. The composition technique defines how different components are actually linked to each other in order to construct a useful program. The composition language is the language in which the connections between the components are made explicit by certain operators.

Our aim is to apply the techniques used in software engineering and bring them into the world of the Semantic Web and its declarative languages. As the Semantic Web languages mature, become widely spread and are used in well-developed complex systems, there will be a need for reuse and composition frameworks. Examples of such Semantic Web languages include the Web Ontology Language OWL [16] and the XML query language Xcerpt [10]. For example, there will be a need to engineer large OWL ontologies from smaller and better understood ontologies also written in OWL. Also, to allow for better reuse of code in Xcerpt programs, which goes beyond the built-in reuse concept of Xcerpt *rules*.

To be able to compose specifications and programs of these languages we need to describe component models for them. As mentioned, this is an essential part of any composition system. One possibility is to write these component models by hand, but this is tedious and error prone. Deliverable I3-D1 [13] presented an approach for automatically deriving a component model for any language given a *meta-model* (specification of the constructs) of that language. Here we describe a tool that automatically derives a component model for a language given its description in OWL. The approach is not limited to any specific language, however, we are specifically concerned with languages relevant for the Semantic Web.

The derived component models are to be used together with *invasive software composition* [18]. They can thus be employed for generic programming, connector-based programming, view-based programming and aspect-oriented programming.

Thus, the purpose of this deliverable is to demonstrate a prototype, named HoMoGen[1], for automatically deriving the essential parts of a component model for any language given its description in OWL. By the essential parts of the component model, we mean the part of the component model that must be specific for every language.

[1] HoMoGen is short for Hook Model Generator

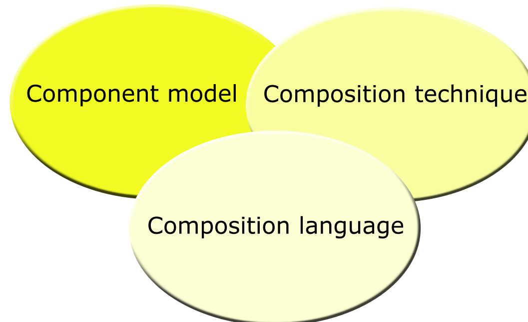

Figure 1: A composition system is comprised from three distinct parts: a component model, a composition technique and a composition language

2 Background

We are dealing with deriving component models for languages, which have a natural place in a software modeling hierarchy. Therefore, in Section 2.1 we first give some background to software modeling abstraction levels in order to make clear at what level in that hierarchy we are working. The generated component models will be used together with invasive software composition. In Section 2.2 we give an introduction to this composition technique. Finally, in Section 2.3, we describe the relation between the component model and the language description, from which it was derived. We also make clear how we get a description of an extended language, in which reuse-aware components will be written.

2.1 Language modeling levels

In this section we briefly summarize the different *modeling levels* which need to be taken into consideration when modeling systems. The modeling levels are organized in a hierarchy as shown in Figure 2. A model at level x is used to describe the modeling constructs used at level $x-1$. We will look at the levels one by one starting at the bottom and moving upwards through the hierarchy.

- *Software objects* At the M0 level we have the software objects which simulate the real world objects that are being modeled.

- *Models* One step up in the hierarchy, at level M1, we have the *software model* level, whose objects are used to describe the software objects in the level below (M0). The objects at this level can be seen as types for the objects at level M0 and are also called *meta-objects*. E.g. the software class *Person* would be used to collect (*type*) all the objects at level M0 that are persons.

- *Meta-models* In the same way, the *meta-model* level M2 introduces types for objects at level M1 and are called *meta-classes* (or *meta-meta-objects*). For example, the meta-meta-object *Class* could be used to describe the specific instance *Person* in level M1, which, in turn, describes all the software objects at level M0 that are persons.

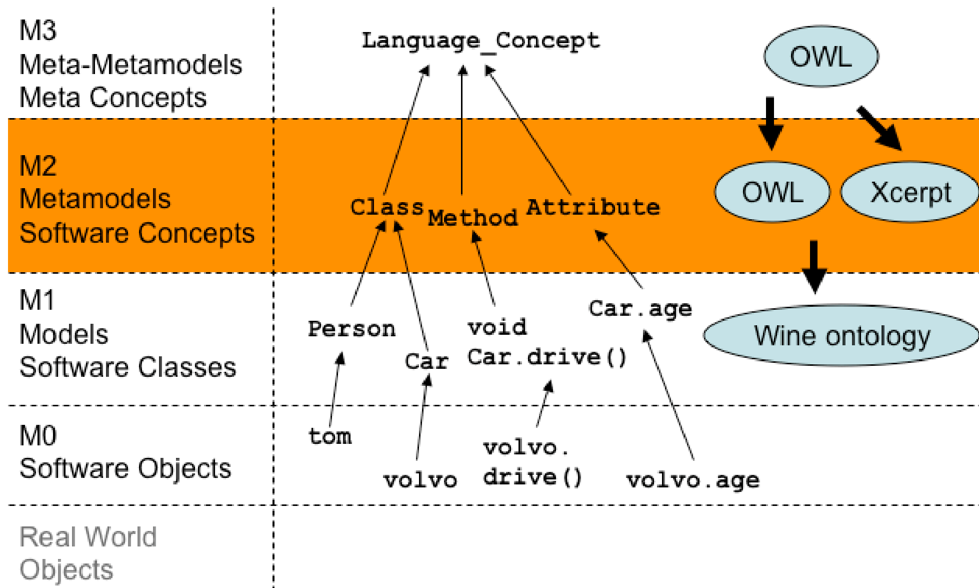

Figure 2: Software modeling hierarchy

What is important to understand at this stage is that a model at a specific level is a language for expressing other models or entities at lower levels in the hierarchy. Specifically, the meta-classes at level M2 are constructs of ordinary programming languages. Therefore, a meta-model can be a description of such a language.

- *Meta-meta-models* Finally in level M3 we have objects that describe the meta-meta-objects of level M2. These objects can describe all concepts that are used in any programming language.

It should be clear that at the meta-meta-model level we can describe specification and programming languages, in particular, ontology languages. A language used for constructing meta-meta-models is a language for specifying other languages. Examples of meta-meta-modeling languages include the MetaObject Facility (MOF) language [2], EBNF [1] and OWL. OWL will be of special importance to our prototype. This is because we assume that the language descriptions we need are described in OWL.

2.2 Invasive software composition

Invasive software composition is an approach towards component based software engineering. In invasive software composition a component is a *set of program fragments*. Therefore, components are referred to as *fragment boxes* to distinguish them from components used with other composition approaches. The reuse abstraction in invasive software composition is called *grey-box* because it makes use of both *white-box* and *black-box* abstraction, as used in traditional software composition. It resembles white-box system, because the actual source code in the components is being modified by composition operators. However, the components are also encapsulated and only accessible through a well-defined interface, as in black-box systems.

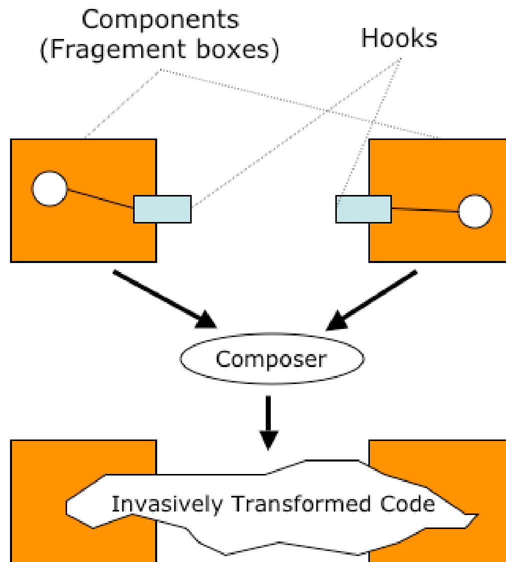

Figure 3: Two components being composed invasively.

The components to be used in invasive software composition have *hooks* defined. A hook is a variation point in a component and declares where and how the component can be modified and reused. A declared hook can be used both for *parameterization* and for *extensibility*. A parameterizable hook functions as a placeholder for other syntactic elements. An extensible hook functions as a variation point where suitable language constructs can be added repeatedly, if necessary. An example of a possible extensible variation point would be the end of a parameter list for a function in e.g. Java.

Hooks can be either *implicit* or *declared*. Implicit hooks are language dependent and are implied by the semantics of the language. For example, every Java method has an entry point and each well-formed XML document has a root element. Thus, we can expect implicitly defined variation points there. The author of a component can also declare hooks explicitly. This will require an extension of the language in question with new keywords to be used for this task. We will define such an extended language of a given language $\mathcal{L}$ in Section 2.3. Furthermore, we will see how these new keywords are automatically derived by our prototype in Section 4.

When finally composing components with hooks invasively, the composition technique adapts the components and transform the hooks with one of the composition operators *bind* or *extend*. This is illustrated in Figure 3. However, this step goes beyond the prototype that we are currently describing. Please refer to [18] for a complete overview of invasive software composition and its techniques.

2.3 Reuse languages and component models

We want to clarify some terminology which will be used when talking about component models. First of all, we refer to the language for which a component model is to be generated, as the *core language*. This is shown in Figure 4. The essential part of a component model for a language

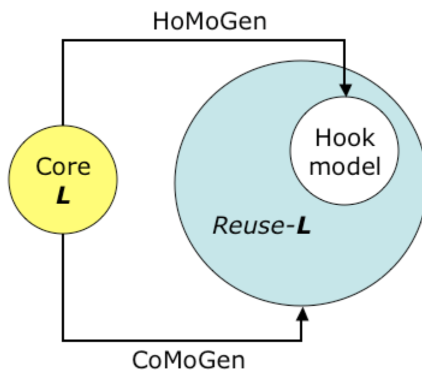

Figure 4: The component model generator, CoMoGen, is a toolset for deriving component models and surrounding tool support for a reuse-language. One part of this toolset is the hook model generator, HoMoGen. A hook model defines language construct to be used as variation points in components written for the core language $\mathcal{L}$.

is made up of its *hook model*. A hook model is a description of the hooks, as defined in Section 2.2, each corresponding to a language construct of the core language. The constructs described in the hook model will be used to declare variation points in components. As seen in Figure 5, the component model is not only made up of the hook model, but also from some additional concepts, which will be used in the naming of components. The collection of these concepts is referred to as *Names* in Figure 5. We should note that this collection of descriptions will not change between component models for different languages, but remains invariant. We will not investigate this further here, instead we focus on what changes between component models for different languages, the hook model. Also, apart from the hook model and the naming concepts, there might be additional information described in the component model. However, we do not discuss this further here.

As seen in Figure 5, the core language, $\mathcal{L}$, is extended, or slightly modified, for a reuse-context. The component model together with the core language extension, constitutes a language, in which re-use aware components of $\mathcal{L}$ will be written. We call this language *Reuse*-$\mathcal{L}$.

3 Deriving a component model

Component models that have been used so far for composing programs with invasive software composition have been hand written. This includes component models for languages such as Java and Prolog that have been used together with the software composition system COMPOST [3]. Our aim is to provide a means to automatically generate a component model for a language given its description as a meta-model. This has the advantage that, as soon as a new language appears, we instantly have access to its component model and do not have to perform the tedious work of hand writing it.

The basic idea of automatically deriving the component model from the language definition comes from the object-oriented language Beta [5]. In Beta, every language construct can be generic, which is achieved by isomorphically mapping each language construct to generic elements of its component model. This is done for the language Beta itself.

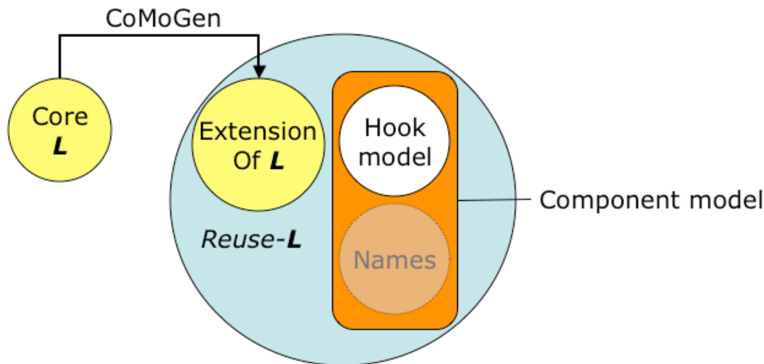

Figure 5: Among other things, CoMoGen will generate a language extension of the core language to be used as part of a more complete reuse-language, Reuse-$\mathcal{L}$, in which reuse-aware components of the core language $\mathcal{L}$ can be written.

We are taking this idea and applying it to any language $\mathcal{L}$ (core language). We derive a component model for $\mathcal{L}$ by first creating a hook model with concepts that are isomorphically derived from the concepts of core language description. The concepts in the hook model are hooks, as described in Section 2.2, which can be used to declare variation points in components. The derivation of a hook model of a language is also illustrated in Figure 6. We note that this horizontal mapping of the core language keeps us on the same meta-model level of the modeling hierarchy. As can be seen in Figure 6, a language construct *Class*, is mapped to the hook *Class_Hook* in the hook model. This convention is used for all constructs when deriving the hook model. The constructs in the hook model allow for both parameterization and extensibility as described in Section 2.2.

We will describe how we generate a hook model for a language in Section 4. There we will assume that the language description is expressed in the Web Ontology Language OWL. The hook model that is generated will also be described in OWL. As noted in Section 2.3, a component model is made up of more information than just a hook model. However, the hook model for a specific language is what makes its component model unique from other component models. As also noted, there are some invariant concept descriptions (e.g. naming), which will be present in every component model. By joining these common descriptions with a hook model for a language, we will have a component model for that language.

4 Prototype: HoMoGen

In this section we will describe the details of our prototype, HoMoGen. In Section 4.1, we first describe common upper ontologies that will be used for every generation of component models from language specifications. Finally, in Section 4.2, we give some technical details of the prototype, specifically the details of how a hook model is generated.

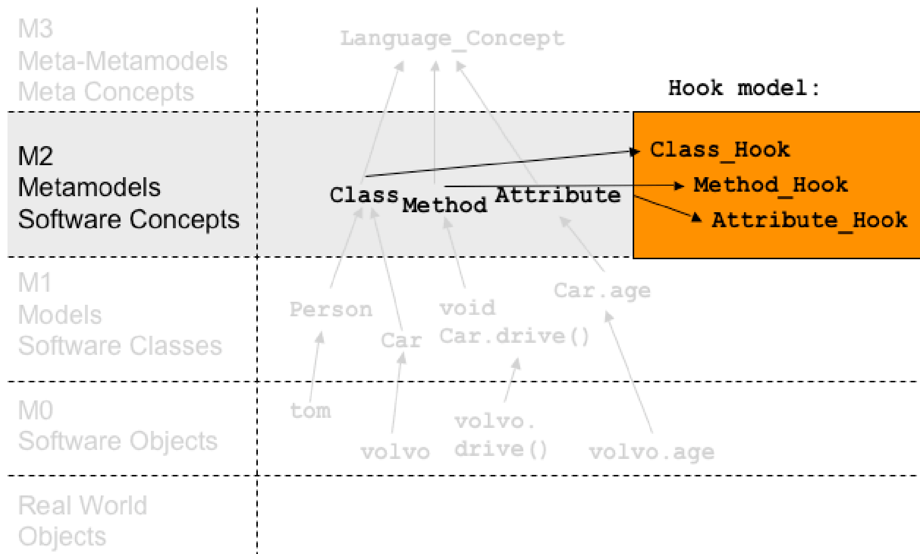

Figure 6: Automatic generation of a hook model is achieved by isomorphically mapping each language construct to a parameterizable and extensible construct in the hook model. This is done on the same software modeling level, i.e. the hook model is a horizontal mapping from the core language meta-model.

4.1 Upper ontologies for language constructs and component models

For the approach of automatically deriving a component model from a language specification we have chosen to make use of two common ontologies. First, we define an *upper ontology for language constructs*. This ontology will define concepts, which are available in any language. For example, *Choices, Aggregates* and *Collections*. More generally, the concept *LanguageConstruct*, which is the top concept in this ontology. Secondly, we define an *upper component model ontology*. This ontology will capture information, which is common to all component models, for example, the concept *LanguageConstructHook*. This ontology references the upper ontology for language constructs because a *LanguageConstructHook*, which will be used in a reuse language (Section 2.3), is also a *LanguageConstruct*.

The relation between the core language, the upper language ontology, the upper component model ontology and the hook model is shown in Figure 7.

One of the purposes of a component model is to describe how components can be connected. Therefore we should therein describe any restrictions made in this regard. Specifically, in the hook model, we describe restrictions on how the hooks can be used for composition. We restrict the values which the hooks can be *bound* to. This is modeled by putting an *allValuesFrom* restriction on the property *boundWith*, which is defined in the upper component model ontology. This particular property is used to restrict the hook model in such a way that the automatically derived hooks can only be bound to their corresponding constructs in the core language description.

For example, suppose we have a concept in the hook model, *Query_Hook*, that has been automatically derived from the concept *Query* in the core language ($\mathcal{L}$) description. Further

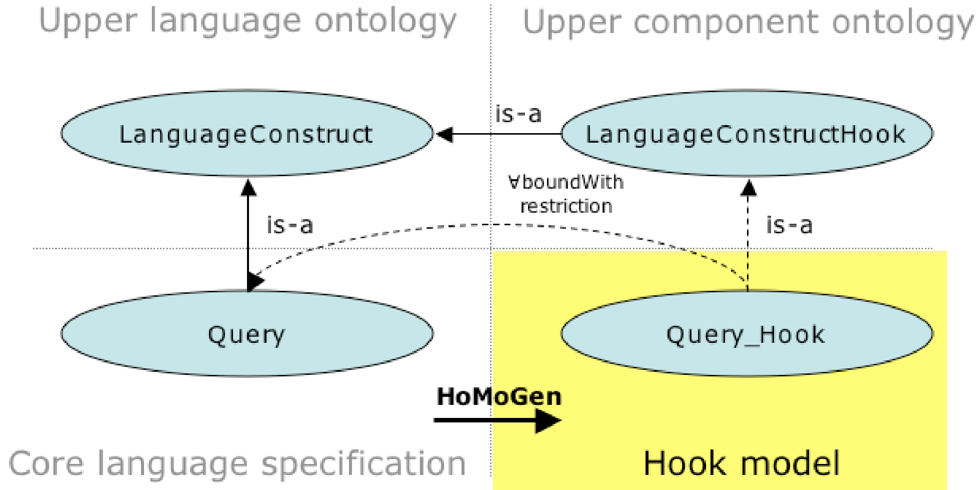

Figure 7: The core language specification and the hook model make use of two upper ontologies. When deriving a hook model for a language, our prototype, HoMoGen, connects the hook model to the upper ontologies. Also, restrictions are put on the derived concepts, using the property *boundWith*. This restricts hooks to be bound with their corresponding constructs of the core language specification.

suppose we have a component, written in Reuse-$\mathcal{L}$, making use of the construct *Query_Hook* to allow a *Query* to be plugged into that specific hook. Thus, the possible language constructs of $\mathcal{L}$ that should be allowed to be bound to the hook *Query_Hook* should be restricted. The restriction should be made to the constructs that belong to the concept *Query* of the core language description. This is done by automatically generating the restriction $\forall boundWith.Query$ for the concept *Query_Hook* in the hook model, see Figure 7. Thus, the hook *Query_Hook* can only be bound to the construct *Query*. This is done for every construct of the core language model when deriving its hook model. The enforcement of this restriction will be made at a later stage when the composition system finally composes components, which we do not deal with yet.

4.2 HoMoGen

A part of the CoMoGen toolset is the hook model generator, HoMoGen, which we will now describe.

HoMoGen is written in Java[2] and makes use of the Java library Jena [4] for handling OWL ontologies. Our prototype deals with reading and creating new models in form of OWL ontologies. Thus, Jena makes handling ontologies much easier, since the library provides primitives for manipulating OWL constructs such as classes and properties. We refer to the web-page[3] of Jena for more information and complete documentation.

The prototype does not have any GUI interface but is simply run from the command line

[2]requires Java 1.5 or higher
[3]http://jena.sourceforge.net/

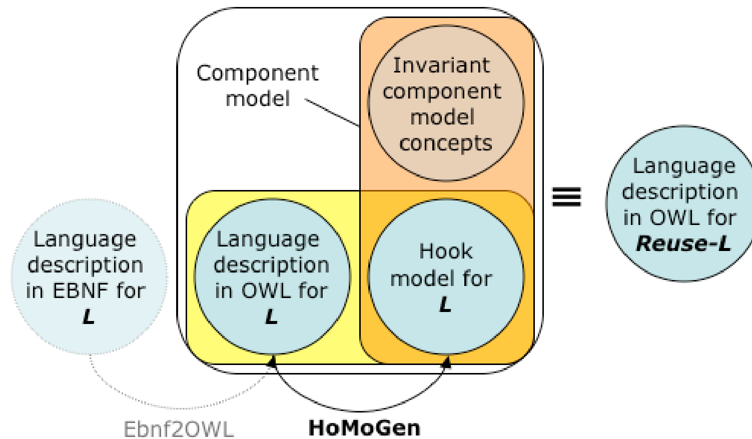

Figure 8: Process of how to derive a component model of a language from its language specification

together with the required arguments. Of course, the prototype can also be run from an IDE like Eclipse[4]. In Figure 9, we can see the prototype being run from Eclipse with no arguments. Then we are informed of the required arguments.

HoMoGen takes two arguments in order to run properly. The first argument is the name of the core language and the second argument is the name of the component model to be generated. The names of these arguments refer to names in a configuration file called *config.xml*, where several pieces of information are declared. For example, below we can see the description of the Xcerpt language specification ontology.

```
<Ontology>
  <Name>Xcerpt</Name>
  <NamespaceURI>
    http://www.owl-ontologies.com/xcerpt.owl
  </NamespaceURI>
  <NamespacePrefix>xcerpt</NamespacePrefix>
  <Location>../resources/xcerpt/xcerpt.owl</Location>
</Ontology>
```

Furthermore, below is information described regarding the Xcerpt component model ontology.

```
<Ontology>
  <Name>XcerptComponentModel</Name>
  <NamespaceURI>
    http://www.owl-ontologies.com/xcerpt_componentmodel.owl
  </NamespaceURI>
  <NamespacePrefix>xcerptcm</NamespacePrefix>
  <Location>
```

[4]http://www.eclipse.org

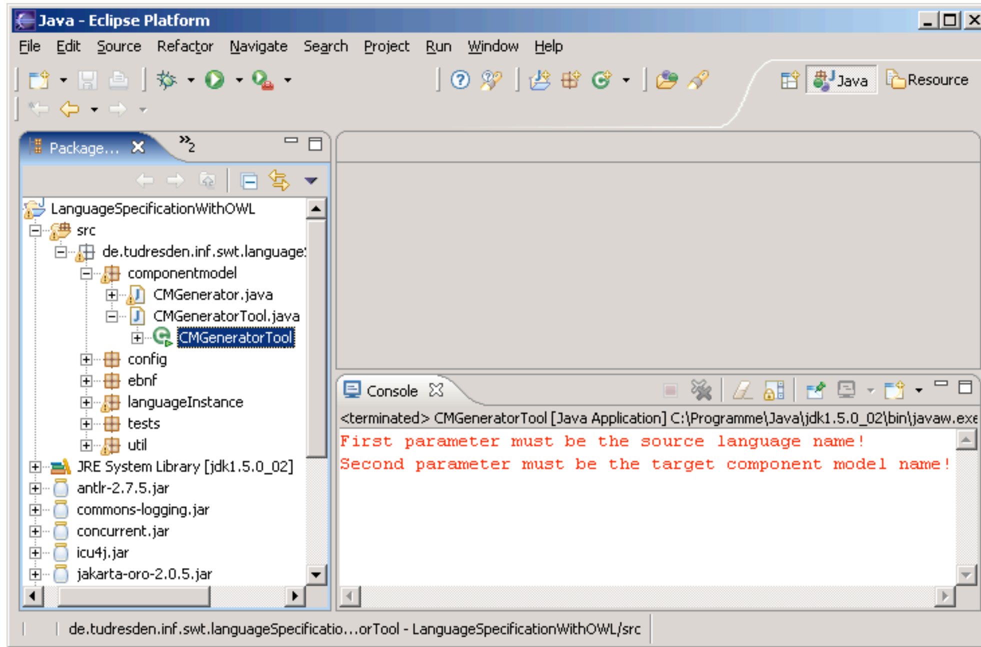

Figure 9: Running the prototype from Eclipse with no arguments

```
    ../resources/xcerpt/xcerpt_componentmodel.owl
  </Location>
</Ontology>
```

Both these pieces of information would be described in a *config.xml* file. Information typically stored in the configuration file is a name for the ontology, a namespace URI, a namespace prefix and the file location where the ontology physically exists. To properly run the prototype and generate a hook model for Xcerpt we pass it the arguments *Xcerpt* and *XcerptComponentModel*. Doing so, we are referring to the information in the *config.xml* file. This can be seen in Figure 10. The first argument specifies the core language description and the second argument specifies the component model to be generated.

Our prototype assumes that there is an OWL description of a core language. However, we want to note that we can also use a tool, Ebnf2OWL, to convert an EBNF-like description of a language to OWL. Both EBNF and OWL are meta-meta-modeling languages, i.e. languages, where it is possible to describe other languages. It can often be easier to describe a language in EBNF notation instead of the more cumbersome XML/RDF notation of OWL. Furthermore, many language are already described in EBNF. Thus, Ebnf2OWL allows us to have a language description in OWL, if it exists in EBNF. This is schematically shown in Figure 8. However, for the prototype we assume the existence of a language specification in OWL. Describing Ebnf2OWL is out of the scope of this report and we will not discuss it further.

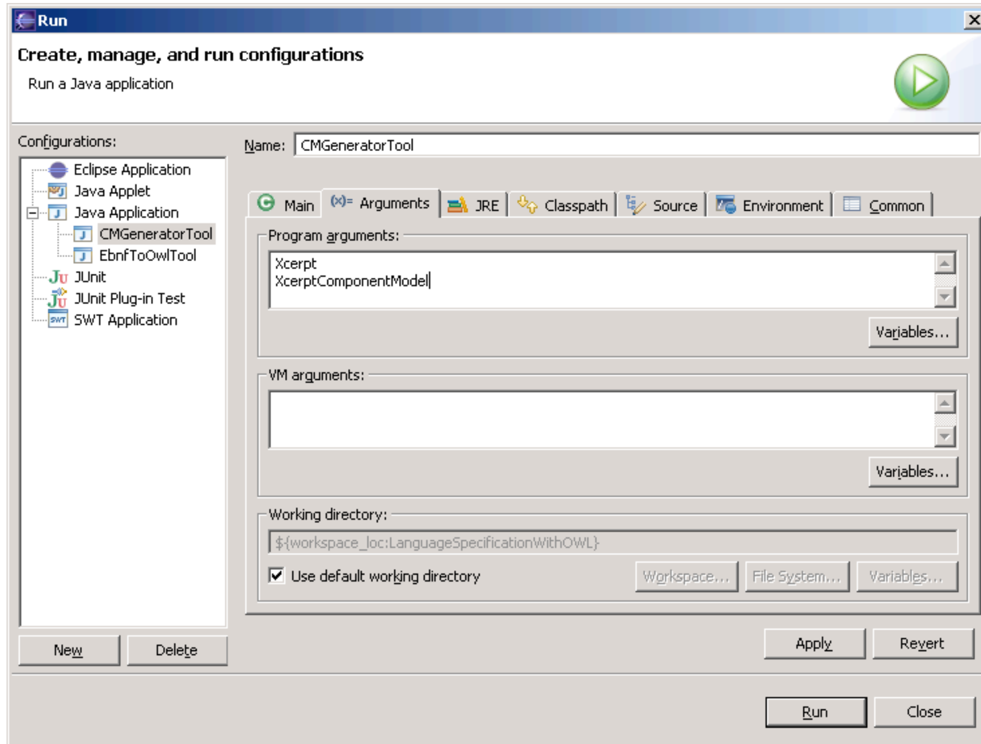

Figure 10: Preparing the prototype to generate a component model for Xcerpt

4.2.1 Deriving generic concepts

We assume we have a meta-model (language description) of a core language $\mathcal{L}$, for which we want to generate a component model. The concepts of $\mathcal{L}$'s meta-model are descriptions of the language constructs of $\mathcal{L}$. Our prototype derive concepts from $\mathcal{L}$'s meta-model in order to generate its hook model. The hook model, together with other invariant descriptions (e.g. for naming, see Section 2.3), constitutes the component model for $\mathcal{L}$.

We will now describe, step by step, how each concept of a hook model is derived from the meta-model of a core language $\mathcal{L}$. We note that when dealing with OWL ontologies we usually use the term *class* to denote a concept. Thus, for each named class $\mathcal{C}$ in the meta-model ontology of $\mathcal{L}$, we take the following steps in creating its hook model.

1. **Create hook concept** Create a class named $\mathcal{C}_Hook$, which is used to denote the hook corresponding to the concept $\mathcal{C}$ in the core language meta-model.

2. **Connect hook concept to upper component model ontology** Make class $\mathcal{C}_Hook$ subclass of the upper component model ontology concept *LanguageConstructHook*.

3. **A hook can appear anywhere its corresponding core construct can** In the meta-model of the core language, the abstract syntax of the language is described, i.e. how certain constructs are allowed to be used. For example, it might be stated that a Java class may be made up of variable declarations and method declarations and definitions.

We want the corresponding hooks of these constructs to be allowed to appear in the same places as the core constructs. For example, a variable declaration hook should be allowed to appear in a class of a Java component, written in Reuse-Java. It is therefore natural to make the hook constructs sub-concepts of the corresponding concepts in the core language extension (see Figure 5). We do this by stating that $\mathcal{C}_Hook$ subClassOf $\mathcal{C}$.

4. **Restrict values to be bound with the hook** Use the property *boundWith*, defined in the upper component model ontology, to restrict the values, which can be bound to the hook. Add an *allValuesFrom* restriction to the class $\mathcal{C}_Hook$ on the *boundWith* property. The restriction should be made to the class $\mathcal{C}$ of the extended core language description. I.e. $\mathcal{C}_Hook$ is restricted with $\forall boundWith.\mathcal{C}$.

5 Examples

In Section 5.1 we describe small pieces of the generation of a hook model for Xcerpt. We also show a small example of components making use of the generated hook model. In Section 5.2 we do the same for OWL.

5.1 Component model for Xcerpt

It is not possible to show the whole hook model for Xcerpt because of space limitations. However, we still want to show in principle what happens with every language construct description found in the core language description when deriving its hook model.

In this example we will look at the Xcerpt construct *Query*. In Figure 11, we see how the description of the *Query* construct is used to derive the corresponding concept, *Query_Hook*, in the hook model.

The following shows a shortened version of the XML/RDF serialization of the concept *Query*, described in OWL, and its corresponding concept in the hook model. The following namespace prefixes are used: *ucmo* refers to the upper component model ontology, *uoflc* refers to the upper ontology for language constructs and *xcerpt* refers to the core language description of Xcerpt.

```
<owl:Class rdf:about="#Query">
  <owl:equivalentClass>
    <owl:Class>
      <owl:unionOf rdf:parseType="Collection">
        <owl:Class rdf:about="#CompoundQuery" />
        <owl:Class rdf:about="#QueryAtom" />
      </owl:unionOf>
    </owl:Class>
  </owl:equivalentClass>
  <rdfs:subClassOf rdf:resource="uoflc:Aggregate" />
  <rdfs:subClassOf rdf:resource="uoflc:ChoiceOfNonTerminals" />
  ...
</owl:Class>

<owl:Class rdf:about="#Query_Hook">
  <rdfs:subClassOf rdf:resource="ucmo:LanguageConstructHook" />
  <owl:equivalentClass>
```

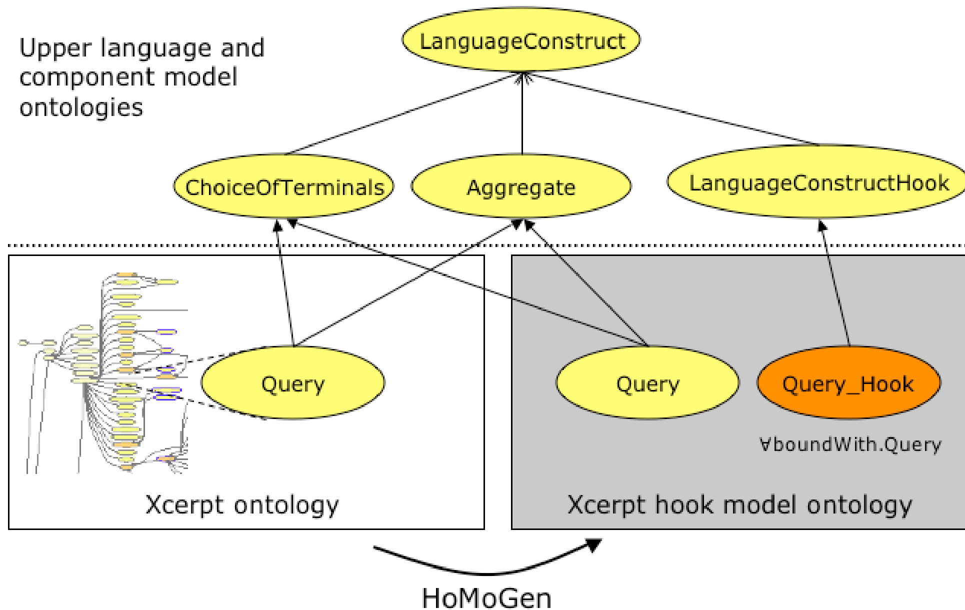

Figure 11: Example of the creation of a hook construct in a hook model from a description of a construct in the core language Xcerpt

```
    <owl:Class>
      <owl:unionOf rdf:parseType="Collection">
        <owl:Class rdf:about="#CompoundQuery_Hook" />
        <owl:Class rdf:about="#QueryAtom_Hook" />
      </owl:unionOf>
    </owl:Class>
  </owl:equivalentClass>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="ucmo:boundWith" />
    <owl:allValuesFrom>
        <owl:Class rdf:about="xcerpt:Query"/>
      </owl:allValuesFrom>
    </owl:Restriction>
  </rdfs:subClassOf>
  ...
</owl:Class>
```

We will look at an example which makes use of the component model for Xcerpt. When declaring rules for querying XML data in Xcerpt, one separates between *query terms* and *construct terms*. I.e. there is a separation between the part of the rule that matches XML data and the part of the rule that constructs the result from the query. An example is shown in [9], where this separation of concerns results in two syntactically very similar rules, for

two semantically similar rules. These queries are shown below. The only difference is in the construct terms (*cons*), where the first rule states *TITLE, all AUTHOR* and the second rules uses *all TITLE, AUTHOR*. The query terms are the same for the two rules.

Listing 1: An Xcerpt rule with a construct term and a query term

```
rule {
  cons {
    result {
      all result {
        TITLE,
        all AUTHOR
      }
    }
  },
  query {
    in { "http://bn.com" },
      bib {{
        book {{
        TITLE —> title ,
        authors {{
          AUTHOR —> author
        }}
      }}
    }}
  }
}
```

Listing 2: The same Xcerpt rule as in Listing 1 except for a slightly different construct term. The query term is the same as in Listing 1, which we want to reuse.

```
rule {
  cons {
    result {
      all result {
        all TITLE,
        AUTHOR
      }
    }
  },
  query {
    in { "http://bn.com" },
      bib {{
        book {{
        TITLE —> title ,
        authors {{
          AUTHOR —> author
        }}
      }}
```

```
    }}
  }
}
```

We intend to allow for the reuse of the query term by writing a component, $c1$, in Reuse-Xcerpt which contains the query term. Furthermore, we write a component, $c2$, which contains one of the construct terms as described above. If the other construct term is to be used, the query term in component $c1$ can be reused for this purpose. The two components, written in pseudo-code, can be found in Listing 3 and Listing 4. The markup "≪" and "≫" is used here to indicate a declared hook in a component.

Listing 3: Component c1, containing the query term, written in Reuse-Xcerpt

```
COMPONENT c1 = {
  query {
    in { "http://bn.com" },
    bib {{
      book {{
        TITLE --> title ,
        authors {{
          AUTHOR --> author
        }}
      }}
    }}
  }
}
```

Listing 4: Component c2, containing one specific construct part, written in Reuse-Xcerpt

```
COMPONENT c2 = {
  rule { cons {
    result {
      all result {
        all TITLE,
        AUTHOR
        }
      }
    },
      <<query_hook_name : Query_Hook>>
    }
  }
```

The components $c1$ and $c2$ could be composed using the following Java pseudo-code:

```
// create the two components
Component queryComp = new Component("c1");
Component ruleComp = new Component("c2");
// locate the declared hook in the 'c2' component
Hook h = ruleComp.findQueryHook("query_hook_name");
// perform the bind operation to connect the components
h.bind(queryComp);
```

The result from this composition would result in the same Xcerpt rule as found in Listing 2.

5.2 Component model for OWL

One of the most powerful composition operators for classes is *parameterized inheritance*. This operator, also called GenVoca operator, has been discovered by Batory and applied in many variants of object-oriented software, e.g. the modelling of product variants in product lines [11]. In essence, this operator parameterizes the superclass of a class, i.e. takes a generic class as input, in which the generic hook denotes a superclass. Such a template is a partial class whose superclass can be determined from outside, e.g. during composition of more complex class hierarchies. Therefore, the operator can be used to combine complex, layered hierarchies of class-based systems. Batory's composition style applies the GenVoca operator between abstraction layers of a system. A variant of the system is composed out of one variant of every layer, glued together by the GenVoca operator. Batory has argued that this architectural style is very useful in building complex systems, such as databases [8], libraries [7], frameworks [6], and other object-oriented software.

Parameterized inheritance combines parameterization (bind a generic hook) and inheritance. Since OWL lacks genericity, it is not possible to realize the GenVoca operator in OWL. However, with a derived component model, it can be realized in Reuse-OWL. What is needed is the concept of a parametrizable superclass reference, derived from a superclass reference in a standard OWL class:

```
<owl:Class rdf:about="#SubClassOf">
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty>
        <owl:ObjectProperty rdf:ID="superDescription"/>
      </owl:onProperty>
      <owl:cardinality rdf:datatype="xsd:int">1</owl:cardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty>
        <owl:ObjectProperty rdf:ID="subDescription"/>
      </owl:onProperty>
      <owl:cardinality rdf:datatype="xsd:int">1</owl:cardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
  ...
</owl:Class>
```

This concept definition will be transformed by HoMoGen to the concept definition of a generic superclass reference:

```
<owl:Class rdf:about="#SubClassOf_Hook">
  <rdfs:subClassOf rdf:resource="ucmo:LanguageConstructHook"/>
  <rdfs:subClassOf>
    <owl:Restriction>
```

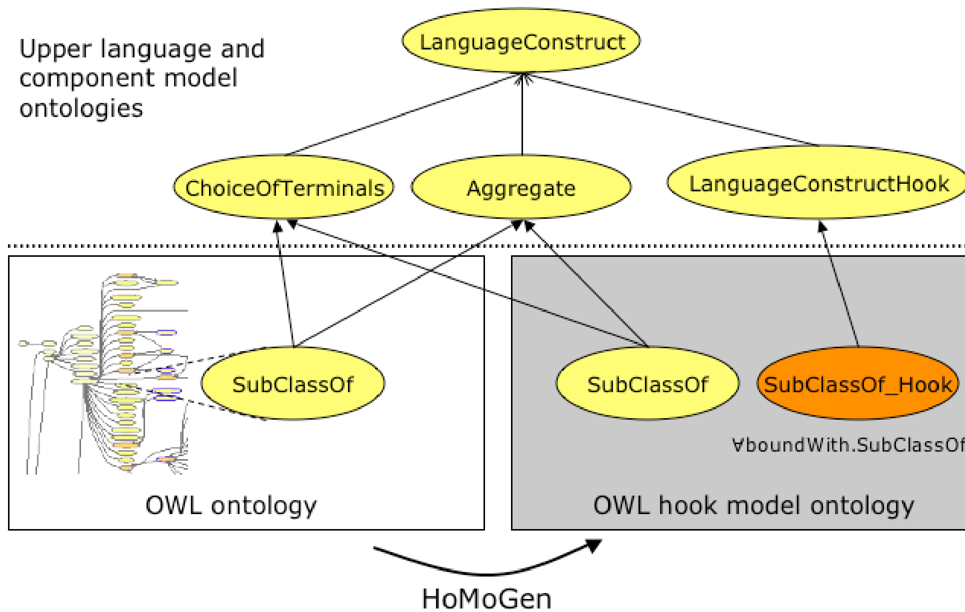

Figure 12: Example of the creation of a hook construct in a hook model from a description of a construct in the core language OWL

```
      <owl:allValuesFrom>
        <owl:Class rdf:about="owl:SubClassOf"/>
      </owl:allValuesFrom>
      <owl:onProperty rdf:resource="ucmo:boundWith"/>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

And this hook can be bound by a parameterization operation, which in this context corresponds to the GenVoca operator.

Consider the following example. A product line of ontologies for living beings should be offered that contains the concepts of parents (father and mother), both for animals and humans. Instead of using multiple inheritance (which would be also be possible, but does not allow for easy extension with new concepts, such as brother or sister), the GenVoca operator is applied.

```
<owl:Class rdf:about="#Man">
  <<super_hook : SubClassOf_Hook>>
</owl:Class>

<owl:DatatypeProperty rdf:ID="hasAge">
  <rdfs:domain rdf:resource="#Man" />
  <rdfs:range  rdf:resource="xsd:positiveInteger"/>
</owl:DatatypeProperty>
```

```
<owl:Class rdf:about="#Woman">
  <<super_hook : SubClassOf_Hook>>
</owl:Class>

<owl:DatatypeProperty rdf:ID="hasHeight">
  <rdfs:domain rdf:resource="#Woman" />
  <rdfs:range  rdf:resource="xsd:positiveInteger"/>
</owl:DatatypeProperty>
```

The second layer of classes is made up of classes that determine the nature of the living being:

```
<owl:DatatypeProperty rdf:ID="hasName">
  <rdfs:domain rdf:resource="#Human" />
  <rdfs:range  rdf:resource="xsd:String" />
</owl:DatatypeProperty>

<owl:Class rdf:about="#Human" />
<owl:Class rdf:about="#Animal" />
```

A class can be composed by applying a class template A of level 0 to a class B of level 1, binding its generic parameter with B. Assuming a `bind` operator which parameterizes a hook with a value, the appropriate composition expressions are:

```
HumanMan = bind(Man.super_hook, Human).
AnimalWoman = bind(Woman.super_hook, Animal).
```

Or, written in the standard notation for parameterization, in which the bind operator is expressed by function parameter binding:

```
HumanMan = Man(Human).
AnimalWoman = Woman(Animal).
```

The result, the two composed classes, have some newly created data-type properties.

```
<owl:DatatypeProperty rdf:ID="hasAge">
  <rdfs:domain rdf:resource="#HumanMan" />
  <rdfs:range  rdf:resource="xsd:positiveInteger"/>
</owl:DatatypeProperty>
<owl:DatatypeProperty rdf:ID="hasName">
  <rdfs:domain rdf:resource="#HumanMan" />
  <rdfs:range  rdf:resource="xsd:String"/>
</owl:DatatypeProperty>

<owl:Class rdf:about="#HumanMan">
  <<super_hook : SubClassOf_Hook>>
</owl:Class>

<owl:DatatypeProperty rdf:ID="hasHeight">
  <rdfs:domain rdf:resource="#AnimalWoman" />
  <rdfs:range  rdf:resource="xsd:positiveInteger"/>
```

```
</owl:DatatypeProperty>

<owl:Class rdf:about="#AnimalWoman">
  <<super_hook : SubClassOf_Hook>>
</owl:Class>
```

Compared with inheritance, the GenVoca operator has the advantage that inheritance expressions (*subclassOf* expressions) are not encoded into the classes, but added by the compositions. This makes specifications much more extensible, since it is very easy to add new alternatives to a layer and achieve new combinations. For instance, if we add a class Brother to level 0 and a class Fish to level 1, classes such as FishMother or HumanBrother can easily be defined, without editing the existing definitions.

Ontology languages that do not provide genericity cannot be used in GenVoca architectural style for ontological frameworks. CoMoGen generates a component model with generic hooks for any XML-based language, delivering the GenVoca operation and its composition style for free.

6 Conclusions

CoMoGen is a generator for component models of arbitrary languages. One part of CoMoGen is the tool HoMoGen, which generates a hook model for the component model. Fed by a language description, a meta-model, it generates a hook model of generic and extensible constructs. The extended language, a combination of the component model and an extension of the core language description (to be generated by CoMoGen) is a *reuse language* and can be used for invasive composition. In this report, we have demonstrated, with examples of OWL and Xcerpt, how the CoMoGen tool can be applied to ontology languages. Not only that specifications can be broken into manageable pieces by generic fragments, but also that powerful composition operators on classes, such as the GenVoca operator, can be realized for any language. Hence, CoMoGen is a decisive tool to realize the principles of invasive software and specification composition for arbitrary languages, in particular ontology languages. This paves the way for a novel ontology, query, and service composition technology, i.e. a fragment-based reuse technology for the Semantic Web.

7 Future work

Several works remain to be done. The prototype CoMoGen must be matured. At the moment, the tool can generate most parts of component models, specifically, the hook model. However, CoMoGen must be extended to also generate type checkers for compositions. Since the tool can be applied to arbitrary language specifications, more case studies should be performed. One of the most important one will be to apply it to the visual ontology language of the REWERSE working group I1 [12]. Other applications of invasive composition, such as view-based development or aspect-oriented development, need to be investigated. Specifically, how they can be used for ontology engineering. Finally, since CoMoGen realizes a simple mapping between meta-models, it raises the question whether other meta-model mappings can be exploited for further reuse concepts.

7.1 Problems

At the moment, the CoMoGen tool can generate the essential parts of component models, but must be extended to generate type checkers for compositions.

If a core language is in an XML-based format, i.e., in an abstract syntax format, fragments of the core and hook language can easily be mixed, i.e. the reuse language is again homogeneously in XML format. It is an open question how to systematically derive parsers for languages in concrete syntax, such that the construction of a hook language becomes simple.

Acknowledgement

This research has been co-funded by the European Commission and by the Swiss Federal Office for Education and Science within the 6th Framework Programme project REWERSE number 506779 (cf. `http://rewerse.net`).

References

[1] Extended BNF. ISO Standard, 13 August 2001. Available at `http://www.iso.org/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=26153`.

[2] MetaObject Facility (MOF) Specification Version 1.4. OMG Specification, April 2002. Available at `http://www.omg.org/docs/formal/02-04-03.pdf`.

[3] COMPOST - the software COMPOsition SysTem. WWW, September 2005. Available at `http://www.the-compost-system.org`.

[4] Jena Semantic Web Framework. WWW Page, 18 August 2005. Available at `http://jena.sourceforge.net/`.

[5] The Beta Language. WWW, August 2005. Available at `http://www.daimi.au.dk/~beta/`.

[6] D. Batory, R. Cardone, and Y. Smaragdakis. Object-oriented frameworks and product lines. In P. Donohoe, editor, *Proceedings of the First Software Product Line Conference*, pages 227–247, Aug. 2000.

[7] D. Batory, V. Singhal, M. Sirkin, and J. Thomas. Scalable Software Libraries. In *Proceedings of the ACM SIGSOFT '93 Symposium on the Foundations of Software Engineering*, pages 191–199. ACM Press, New York, Dec. 1993.

[8] D. Batory, V. Singhal, J. Thomas, S. Dasari, B. Geraci, and M. Sirkin. The GenVoca model of software-system generation. *IEEE Software*, 11(5):89–94, Sept. 1994.

[9] F. Bry and S. Schaffert. A gentle introduction into xcerpt, a rule-based query and transformation language for xml. In *Proceedings of International Workshop on Rule Markup Languages for Business Rules on the Semantic Web, Sardinia, Italy (14th June 2002)*, 2002.

[10] F. Bry and S. Schaffert. The xml query language xcerpt: Design principles, examples, and semantics. In *Revised Papers from the NODe 2002 Web and Database-Related Workshops on Web, Web-Services, and Database Systems*, pages 295–310, London, UK, 2003. Springer-Verlag.

[11] Don Batory and Sean O'Malley. The Design and Implementation of Hierarchical Software Systems with Reusable Components. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 1(4):355–398, Oct. 1992.

[12] Grigoris Antoniou et. al. A First-Version Visual Rule Language. Technical report, Eindhoven University of Technology, 2004.

[13] Ilie Savga, Charlie Abela, Uwe Assmann. Report on the design of component model and composition technology for the Datalog and Prolog variants of the REWERSE languages. Technical report, Technical University of Dresden, 2004.

[14] M. D. McIlroy. Mass-produced software components. In J. M. Buxton, P. Naur, and B. Randell, editors, *Software Engineering Concepts and Techniques (1968 NATO Conference of Software Engineering)*, pages 88–98. NATO Science Committee, Brussels, Oct. 1969.

[15] O. Nierstrasz and T. D. Meijler. Research directions in software composition. *ACM Computing Surveys*, 27(2):262–264, June 1995.

[16] P. F. Patel-Schneider, P. Hayes, and I. Horrocks. OWL web ontology language semantics and abstract syntax. W3C Recommendation, 10 February 2004. Available at `http://www.w3.org/TR/owl-semantics/`.

[17] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, New York, 1998.

[18] Uwe Assmann. *Invasive Software Composition*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2003.