I3-D7

Prototypical composition ontology for rule-based 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:	R (report)
Dissemination level:	PP (restricted to FP6 participants)
Document number:	IST506779/Dresden/I3-D7/D/PP/a1
Responsible editors:	Jakob Henriksson
Reviewers:	S. Lukichev (BTU Cottbus)
Contributing participants:	Dresden, Malta, Nancy
Contributing workpackages:	I3
Contractual date of deliverable:	28 February 2006
Actual submission date:	April 21, 2006

Abstract

This report presents an ontology for the composition of rule-based ontology languages. Since typing compositions is an important issue, the ontology consists of components: two upper-level ontologies, the metamodel of the ontology language, and a metamodel of reuse constructs that play an important role in composition. With the interplay of these components, type-safe composition of ontology components can be achieved.

Keyword List

component-based ontology engineering, composition ontology, 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.

Prototypical composition ontology for rule-based languages

Uwe Aßmann[1], Jakob Henriksson[1], Ilie Savga[1], Matthew Montebello[2], Claude Kirchner[3]

[1] Faculty of Informatics, Technical University of Dresden
Email: {uwe.assmann|jakob.henriksson|ilie.savga}@tu-dresden.de
[2] Department of Computing and IT, Junior College, University of Malta
Email: mmont@cs.um.edu.mt
[3] LORIA, Nantes
Email: Claude.Kirchner@loria.fr

April 21, 2006

Abstract
This report presents an ontology for the composition of rule-based ontology languages. Since typing compositions is an important issue, the ontology consists of components: two upper-level ontologies, the metamodel of the ontology language, and a metamodel of reuse constructs that play an important role in composition. With the interplay of these components, type-safe composition of ontology components can be achieved.

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

Contents

1 Preliminaries

This paper presents an ontology for composition of rule-based ontology languages. Since typing compositions is an important issue, the ontology consists of several components: two upper-level ontologies, the metamodel of the ontology language, and a language-specific metamodel of reuse constructs that play an important role in composition. With the interplay of these metamodel components, type-safe composition of ontology components can be achieved.

Actually, seen in more detail, not one single ontology for ontology composition is given; instead, for every involved language a *language-specific composition ontology* is generated. As an advantage, this language-specific ontology provides types for a type-safe composition of ontology components. With a generic ontology, this could not easily be achieved, because typing constraints cannot easily be unfolded into explicit inheritance relationships between hooks and the values to be substituted. With a composition ontology specific to a rule-based language, however, typing constraints can be expressed in a general typing rule and inheritance constraints between the language-specific components and the upper-level ontologies. In this way, composition can easily be type checked in a safe manner using an ontology reasoner.

Since the presented ontology is the basis for the component-model generator CoMoGen, which is developed as a demonstrator action in REWERSE, some of the material has already been presented in the prototype description of CoMoGen, i.e., deliverable I3-D5 [13]. However, here, more details on the upper-level ontologies are explained, in particular their relationship to the typing of the compositions. Hence, Sections 2–4.1 are taken up again from deliverable I3-D5 and refined. Section 4 gives then a more detailed overview on the composition ontologies and explains the roles of their components in the typing of compositions. Section 5 applies the technology to the query language Xcerpt. Finally, Section 6 wraps up and articulates future work.

2 Introduction

Developing software from smaller existing parts, called *component-based development* [11], has been around for a long time in the software engineering community [8]. 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 [9]. Among other things, creating software from components allows 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 connecting 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

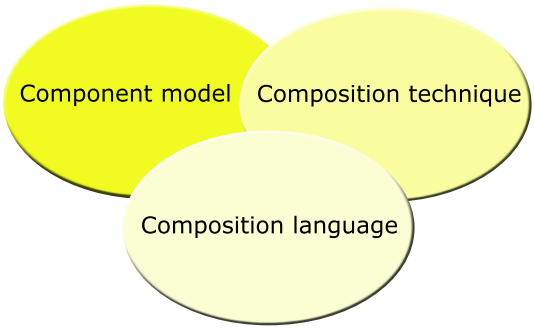

Figure 1: A composition system is comprised from three distinct parts: a component model, a composition technique and a composition language

be a need for reuse and composition frameworks. Examples of such Semantic Web languages include the Web Ontology Language OWL [10] and the XML query and transformation language Xcerpt [6]. For example, there will be a need to engineer large OWL ontologies from smaller and better understood ontologies also written in OWL. Also, composition allows for better reuse in Xcerpt programs, which goes beyond the built-in reuse concept of Xcerpt, *rules*.

In contrast to the standard situation in software engineering, ontology engineers do not need binary component models [11], but source code components models. The question of binary formats is not important, instead, there is a need to reuse ontology fragments in flexible ways. Luckily, such *fragment components (fragment boxes)* are offered by a technology called *invasive software composition* [12]. Fragment components are generic, i.e., can be adapted by parameterization, and are also easily extensible, i.e., are amenable for view-based and aspect-oriented modeling. Clearly, these features are also interesting for ontology languages.

Hence, to be able to compose specifications and programs of Semantic Web languages, we need invasive component models for them. One possibility is to write these component models by hand, but this process, as it can be seen from the component model of Java [12], is laborious. Hence, deliverable I3-D1 [7] presented an approach for automatically deriving a component model for any language given a *meta-model* (specification of the constructs) of that language. This approach makes it much more easy to create invasive component models for new languages, also language families, such as the family of Semantic Web ontology languages. Here, we generalize this approach to a *composition ontology*. This composition ontology contains a description of a (core) language in OWL, i.e., a language ontology, the derived component model for the language, as well as two upper-level ontologies that factor out important language-independent and component-model-independent aspects. The approach is not limited to any specific language, however, here we are specifically concerned with languages relevant for the Semantic Web.

Thus, the purpose of this deliverable is to explain the components of the composition ontology. We argue that a componentization is important to on the one hand reuse knowledge of upper-level ontologies for all languages, and on the other hand, to use the concrete concepts in the lower-level ontologies for better typing of the compositions.

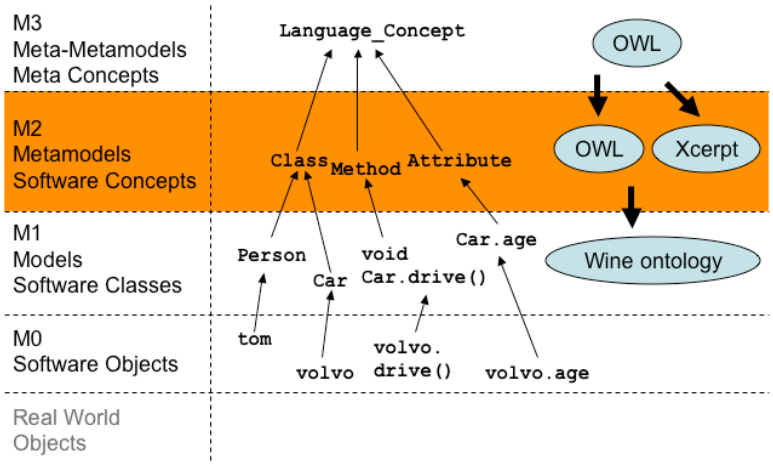

Figure 2: Software modeling hierarchy

3 Background

We are dealing with deriving component models for languages, which have a natural place in a software modeling hierarchy. Therefore, in Section 3.1 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 3.2, we give an introduction to this composition technique. Finally, in Section 4, 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.

3.1 Language modeling levels

In this section, we briefly summarize the different *modeling levels*, which need to be taken into consideration when modeling systems [2] 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-

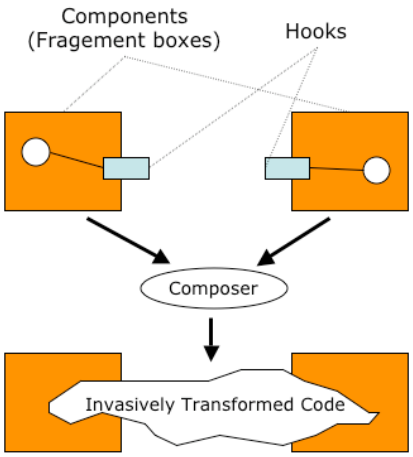

Figure 3: Two components being composed invasively.

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.

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 or modeling 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 or modeling language.

It should be clear that at the meta-meta-model level we can describe programming or modeling languages, in particular, ontology languages. A language used for constructing meta-meta-models is a language for specifying other languages and is usually called a *language description language* or *metalanguage*. In this paper, we employ an ontology language as a metalanguage on M3. To model metamodels of ontology languages on M2 seems to be natural, in particular because such a language provides reuse concepts such as inheritance, rich relationships and a description logic to model constraints of the static language semantics. More specifically, we use OWL and SWRL as metalanguages, although in general, other logic languages can be used. Other examples of metalanguages include the MetaObject Facility (MOF) language with the OCL constraint language [2], or EBNF [1].

3.2 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 components* 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

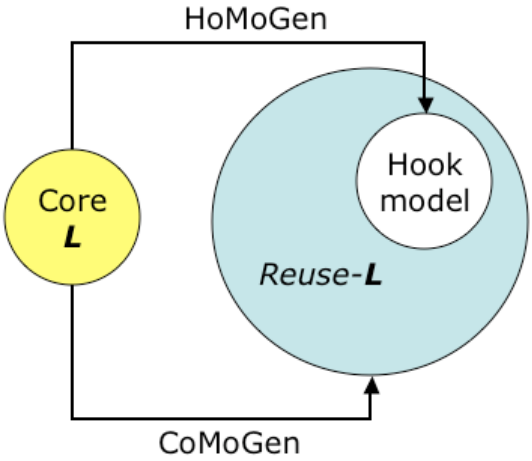

Figure 4: CoMoGen (component model generator), is a toolset for deriving component models and surrounding support for a reuse-language. One part of this toolset is the HoMoGen (hook model generator). A hook model defines language construct to be used to describe variation points in components.

software composition. On the one hand, grey-box components resembles white-box components, because the actual source code in the components is being modified by composition operators. On the other hand, the components are encapsulated and only accessible through a well-defined interface, a *composition interface*.

The composition interface of a grey-box component defines *hooks*, variation points telling where and how the component can be modified for reuse. A hook can be used both for *parameterization* and for *extension*. A parameterizable hook functions as a placeholder for other syntactic elements. An extensible hook serves as an extension point where suitable language constructs can be added repeatedly, if necessary. (An example of an extension point would be the end of a parameter list of a function.)

Hooks can be either *implicit* or *declared*. Implicit hooks depend on the language and are implicitly defined by the semantics of the language. For example, since every Java method has an entry point, it can be extended by a composition operator prefixing prologue code to the entry point. The author of a component can also declare hooks explicitly. Such hook definitions require an extension of the language in question. We will define such an extended language of a given language $\mathcal{L}$ in Section 4.

Finally, when composing fragment components invasively, the composition technique adapts them, transforming the hooks with one of the composition operators *bind* (parameterization) or *extend*. This is illustrated in Figure 3. Please refer to [12] for a complete overview of invasive software composition and its techniques.

4 A Composition Ontology in Compartments

As said above, a composition system needs a component model (which describes the components), a composition technique (which will define basic composition operations), and a compo-

sition language (which glues together components in the large). Hence, a general composition ontology that should describe valid compositions in composition systems, must take all these ingredients into account.

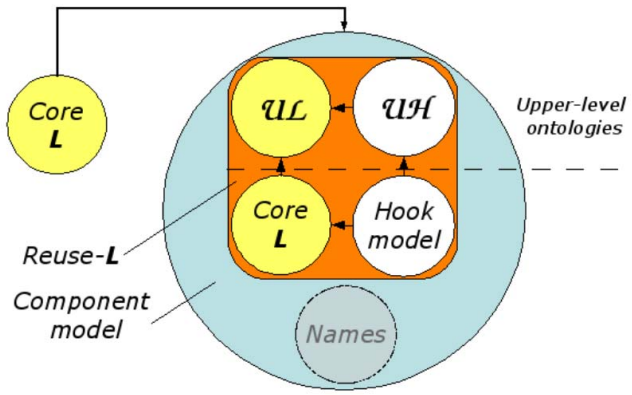

Figure 5: The reuse-language is made up of 4 compartments: one language meta-model, one generated hook-model and two upper-level ontologies, one for language and one for hook models. Additionally, some other concepts are included to complete the component model, e.g. concepts for naming of components

As a start, we want to clarify some terminology for the rest of the paper. 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 variable part of each language's component model is made up of its *hook model*. It contains hook constructs, each of which corresponds to a language construct of the core language. The *component model*, called $\mathcal{CM}(\mathcal{L})$, consists of the hook model $\mathcal{HM}(\mathcal{L})$ and some other parts, like naming concepts, that will be reused for all component models. Together, core language and component model describe an enriched language, in which re-use aware components of $\mathcal{L}$ can be written. We call this full language *Reuse-*$\mathcal{L}$.

Based on this terminology, we unfold the structure of our proposed composition ontology. Mainly, it consists of two layers of ontologies (Figure 5):

Language-specific knowledge This layer contains the meta-model of the core language $\mathcal{L}$ (the language ontology). Then, the layer contains a second ontology $\mathcal{HM}(\mathcal{L})$, the hook model, which is derived from the core language ontology, providing reuse concepts for the core language. $\mathcal{HM}(\mathcal{L})$ also relates reuse-concepts to the concepts of the core language ontology describing $\mathcal{L}$.

Language-neutral knowledge To factor out knowledge for both ontologies that are language-independent, and to reuse them for other languages, we define a second layer of upper-level ontologies that contain abstract concepts for reuse in the lower layer. We consider an upper ontology for a language $\mathcal{UL}$ and an upper ontology for the component model $\mathcal{UH}$.

All ontology components together form the ontology of the reuse language *Reuse-*$\mathcal{L}$. If $\mathcal{L}$ is an ontology language, *Reuse-*$\mathcal{L}$ is a reuse language for ontologies, i.e., a language for

component-based ontology engineering, which describes fixed parts of ontology components in $\mathcal{L}$, but variable and extensible parts of ontology components in $\mathcal{HM}(\mathcal{L})$.

In the following, first, we describe how an invasive hook model $\mathcal{HM}(\mathcal{L})$ is derived from a core language $\mathcal{L}$. Then, the upper-level ontologies are described.

4.1 Deriving an Invasive Component Model

Component models that have been used so far for composing programs with invasive software composition have been written by hand. 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 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 [4]. In Beta, every language construct can be generic, which is achieved by isomorphically mapping each language construct to generic elements of its component model. However, this is done for the language Beta specifically.

We take this idea and apply 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 the core language description. The concepts in the hook model are hooks, as described in Section 3.2, 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. 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 language constructs of $\mathcal{L}$.

The derived hook model allow not only for parameterization, but also for extensibility. To this end, for each extensible language construct of $\mathcal{L}$, an extensibility hook in $\mathcal{HM}(\mathcal{L})$ is derived. An extensible language construct is a construct that may appear in a list or set of constructs, for instance, a parameter that appears in a parameter list; an import declaration that appears in an import list, etc.

Every generated hook concept must inherit from the original language concept in the core language, because a hook can appear in place of a construct. Besides the hook model, the component model has also other ingredients. Additionally, there are some invariant concept descriptions, which will be present in every component model. For instance, a component that describes naming issues has to be provided for every component model. Also, common upper-level ontologies for languages and hook models are used between component models.

For the implementation of the hook model generator, we assume that languages are described in the Web Ontology Language OWL. The hook model is also generated in OWL. This is particularly apt for the context of the Semantic Web, because OWL is the main ontology language, however, other metalanguages can be employed.

4.2 Upper ontologies for language constructs and component models

In this section, we describe common upper ontologies that will be used for every component model derived from a language specification.

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*

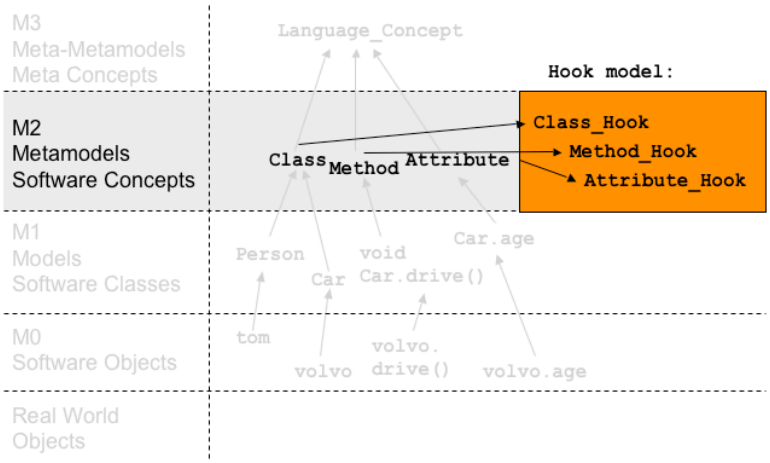

Figure 6: Automatic generation of a hook model is achieved by isomorphically mapping each language construct to a parameterizable and extensible construct in the hook model. This is done on the same software modeling level, i.e., the hook model is a horizontal mapping from the core language meta-model.

language constructs $\mathcal{UL}$. This ontology defines concepts, which are used in any language, for example, *Choices, Aggregates* and *Collections*. Also, it includes the concept *LanguageConstruct* as the top concept. Secondly, we define an *upper component model ontology* $\mathcal{UH}$, which captures information common to all component models, for example, the concepts *ChoiceHook* or *LanguageConstructHook*. The upper component model ontology references the upper ontology for language constructs because a *LanguageConstructHook*, which will be used in a reuse language, 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.

4.2.1 The upper ontology for language constructs

When describing languages, three main types of constructs can be identified: *choice, aggregate* and *collection*.

Choice Describes a construct having a set of different variants (e.g. a boolean operation can either be "or" or "and")

Aggregate Describe constructs that are defined by a set of parts (e.g. a condition is made up of a boolean test, an instruction to be executed if the test holds and an optional instruction to be executed otherwise)

Collection Describe constructs that are made up a variable number of instances of one specified construct (e.g. a declaration list is made up of a list of declarations)

These constructs are described in the upper ontology for language constructs $\mathcal{UL}$ (see Appendix A).

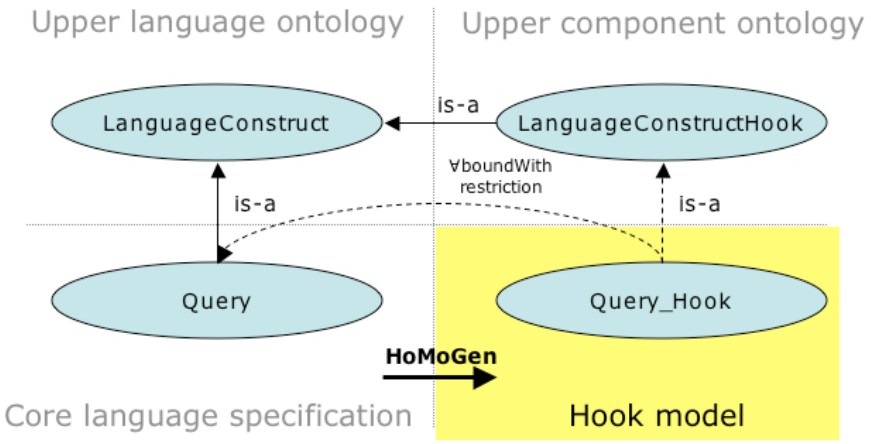

Figure 7: The core language specification and the hook model make use of two upper ontologies. When deriving a hook model for a language, our prototype, HoMoGen, connects the hook model to the upper ontologies. Also, restrictions are put on the derived concepts, using the property *boundWith*. This restricts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

4.2.2 The upper ontology for component model constructs

This ontology $\mathcal{UH}$ contains, among others, the following concepts and relationships (see Appendix B):

LanguageConstructHook *LanguageConstructHook* is a top level concept, which all hooks in the hook models are sub-classes of

boundWith The property *boundWith* is used by the hook model to model the restriction that each hook can only be bound to its corresponding construct in the core language $\mathcal{L}$ or a subclass thereof

extensibleWith The property *extensibleWith* is used by the hook model to restrict extensible hooks in such a way that they can only be extended with their corresponding constructs in the core language $\mathcal{L}$ or a subclass thereof

4.3 Representation of Typing Rules

One of the purposes of a component model is to describe how components can be composed legally. Therefore, a component model should describe constraints or typing rules for compositions. Specifically, the hook model should describe restrictions on how the hooks, both implicit and declared, can be used for composition.

We apply the following generic typing rules in the composition ontology. As an example, we use the language Xcerpt [5] as core language and derive a hook model for it, $\mathcal{HM}(Xcerpt)$, extending Xcerpt to a reuse language Reuse-Xcerpt.

The first rule limits the use of hook constructs to appear in places where their associated construct in $\mathcal{L}$ are allowed to appear.

TR-1 For all $c \in \mathcal{L}$, $\mathcal{HM}(c)$ shall be subclass of c.

In Reuse-$\mathcal{L}$, a hook $\mathcal{HM}(c)$ can appear in any context where its corresponding core construct c can occur. Therefore, in $\mathcal{HM}(\mathcal{L})$, we introduce a statement for making the hook construct a sub-concept of the corresponding concept in the core language (see Figure 5).

For example, it might be stated that an Xcerpt query-construct is to appear in a specific place in an Xcerpt construct-query rule. We want the corresponding hook constructs to be allowed to appear in the same places as the core constructs. For example, we want a query hook to be allowed to appear in a valid place of a construct-query rule in Reuse-Xcerpt. It is therefore natural to make the hook concepts sub-concepts of the corresponding concepts. I.e., to make Xcerpt's construct *Query* parameterizable, a concept *Query_Hook* is created in $\mathcal{HM}(Xcerpt)$, and a constraint *Query_Hook subclassOf Query* is added to $\mathcal{HM}(Xcerpt)$.

TR-2 When binding a variation hook in a fragment component, for all concepts $c \in \mathcal{L}$, a hook typed with variation hook concept $h = \mathcal{HM}(c) \in \mathcal{HM}(\mathcal{L})$ can only be *bound* to fragments of type c or a subclass thereof.

This constraint is modeled by putting an *allValuesFrom* restriction on the property *boundWith*, which is defined in the upper component model ontology. For example, for the Xcerpt hook concept $Query\_Hook$ the following restriction is added to the class $\mathcal{HM}(Query\_Hook)$: $\forall boundWith.Query$.

This property *boundWith* is thus 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.

The next rule repeats the restriction rule for extension hooks.

TR-3 When extending an extension hook in a fragment component, for all concepts $c \in \mathcal{L}$, a hook typed with variation hook concept $h = \mathcal{HM}(c) \in \mathcal{HM}(\mathcal{L})$ can only be *extended* with fragments of type c or a subclass thereof.

This constraint is modeled by putting an *allValuesFrom* restriction on the property *extensibleWith*, 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 extensible hooks can only be bound to their corresponding constructs in the core language description.

4.4 Advantages of the Ontology Compartments

The approach of two layers of the composition ontology, one language-independent, and one language-specific, has several advantages.

First, the language-specific layer, since it is re-generated for each core language, provides simpler typing of the language specific knowledge. All replacement constraints are effectively encoded into subclass relationships between the generated hook model and the core language meta-model (constraint TR-1). Using these subclass constraints, the generic typing rules TR-2 and TR-3 can check whether the right fragments are put into hooks, i.e., whether components are parameterized or extended correctly, with regard to the specified language ontology.

If the re-generation step was not performed, i.e., a single, language-universal composition ontology would be employed, constraints TR-2 and TR-3 would be second-order constraints on TR-1. For such constraints, a second-order reasoner should be employed, but due to the re-generation process, this is not required. The re-generation is possible because the type checking process of fragment-based compositions can be split into two phases: a pre-processing phase, which generates the language-specific layer, and a second, main typing phase, which employs a first-order description logic layer to type-check compositions.

Secondly, the two-level architecture enables the reuse of language-neutral knowledge in upper ontologies. The connection made between the language-specific ontologies and the upper-level ontologies are actually rather simple; it can be replaced by another, language-specific set of constraints that maps language constructs to abstract language constructs, to provide tailored reuse rules for them.

For instance, it is possible to factor out knowledge about uses and definitions into Use and Definition abstract concepts. In this way, use-related features of languages concepts can be separated from definition-related concepts. Clearly, the former can easily benefit to the definition of *Data Manipulation Languages (DML)*, while the latter can benefit to the definition of *Data Definition Languages (DDL)*.

Finally, the two-level architecture can be extended to more layers. For instance, the Use and Definition abstract concepts of above can be factored into a separate *use/def layer*, so that they can be exchanged more easily. It remains future work to see whether this can help to simplify the process to construct ontology language component models.

5 How to Compose Xcerpt

In this section, we will look at two examples dealing with composing program fragements of the XML query and transformation language Xcerpt [6].

The construct-query rule in Listing 1 queries a bibliography file and collects all titles and authors from it and then creates a result (in the construct part, lines 2-7), which lists all authors for each title. We could factor out, for example, the query part and make it into a component, which could possibly be reused for other rules that constructs the result in a different manner. This component can be found in Listing 2. We then have a second component (Listing 3), which contains the construct part. Instead of having a query where expected, a declared query hook can be found in its place (line 9 in Listing 3). By binding the query component (Listing 2) into the hook declared on line 9 in Listing 3 we are able to construct the expected rule as the result, shown in Listing 1.

Listing 1: Result from composing component in Listing 3 with the query component in Listing 2

```
1 CONSTRUCT
2   result {
3     all result {
4       var Title ,
5       all var Author
6     }
7   }
8 FROM
```

```
  in {
    resource { "http://bn.com" } ,
    bib {{
      book {{
        var Title -> title ,
        authors {{
          var Author -> author
        }}
      }}
    }}
  }
END
```

Listing 2: Component2: Xcerpt component describing a query

```
  in {
    resource { "http://bn.com" } ,
    bib {{
      book {{
        var Title -> title ,
        authors {{
          var Author -> author
        }}
      }}
    }}
  }
```

Listing 3: Component1: Xcerpt component describing how to construct an answer from a query

```
CONSTRUCT
  result {
    all result {
      var Title ,
      all var Author
    }
  }
FROM
   bindQuery *bibQuery
END
```

We can then also further factor out other things from this rule. For example, we make use of variables, both in the query part and in the construct part, which could be made into components. These can be found in Listing 4 and in Listing 5.

Listing 4: Component3: A variable called Author

```
var Author
```

Listing 5: Component4: A variable called Title

```
1 var Title
```

To this end we need to modify the query component and declare variable hooks instead of the actualy variables. These hooks will later be bound to the correct variable components we just factored out. The declared hooks in the new query component can be found on lines 5 and 7 in Listing 6.

Listing 6: Component5: Xcerpt component describing a query with two declared hooks (lines 5 and 7)

```
1  in {
2    resource { "http://bn.com" } ,
3    bib {{
4      book {{
5        hookVariable $title -> title ,
6        authors {{
7                hookVariable $author -> author
8        }}
9      }}
10   }}
11 }
```

In this extended example, with 2 additional variable components, the construct component (Listing 7) gets a little more complicated. This in order to facilitate binding the components together. Thus, Reuse-Xcerpt is used also as composition language, specifiying how the Xcerpt components should be bound together into a complete construct-and-query rule. The hooks in the query component (Listing 6) are bound on lines 1 and 2 in Listing 7. When all the hooks are bound as specified, the complete construct-query-rule is created as expected (Listing 1).

Listing 7: Component6: Xcerpt component describing how to construct an answer from a query with two declared hooks (lines 7 and 9)

```
1  bind *bibQuery $title *varTitle1
2  bind *bibQuery $author *varAuthor1
3
4  CONSTRUCT
5    result {
6      all result {
7        bindVariable *varTitle2 ,
8        all
9          bindVariable *varAuthor2
10     }
11   }
12 FROM
13     bindQuery *bibQuery
14 END
```

6 Conclusions

This report has presented a component-based, two-layered approach to a composition ontology. With this approach, not only core languages are described, but a component model as well as composition typing constraints come for free. Invasive composition systems can be constructed for languages easily, on button-press. The resulting concrete ontology is constructed from several components, organized in two layers, of which the lower serves typing purposes for the compositions, and the upper serves reuse of language modelling information.

The approach can be employed to generate invasive component models for all members of the Semantic Web ontology languages. Since with an invasive component model, parametrizable and extensible ontology components can be written, it is a prerequisite for adaptable and view-based component-based ontology engineering.

6.1 Future work

Now, the approach should be thoroughly tested in case studies, e.g., modelling ontologies with mixin layers. One of our next experiments will chainsaw the Gene Ontology into mixin layers, which should render it much easier. At the moment, University of Malta investigates the technology for a component model of OWL-S. Another case study will be a component model for the visual ontology language of REWERSE-I1.

The question remains how the generic technology of this report can be transformed into a proposal for the W3C consortium, proposing a standard typing framework for component-based ontology engineering.

6.2 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. Technology should be developed 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`).

A Appendix A: The Upper Ontology for Language Constructs

```
<?xml version="1.0"?>
  <rdf:RDF
      xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
```

```
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:owl="http://www.w3.org/2002/07/owl#"
  xmlns="http://www.owl-ontologies.com/abstractsyntax.owl#"
xml:base="http://www.owl-ontologies.com/abstractsyntax.owl">
<owl:Ontology rdf:about=""/>
<owl:Class rdf:ID="ChoiceOfNonTerminals">
  <rdfs:subClassOf>
    <owl:Class rdf:ID="LanguageConstruct"/>
  </rdfs:subClassOf>
</owl:Class>
<owl:Class rdf:ID="Aggregate">
  <rdfs:subClassOf rdf:resource="#LanguageConstruct"/>
</owl:Class>
<owl:Class rdf:ID="OrderedCollection">
  <rdfs:subClassOf>
    <owl:Class rdf:ID="Collection"/>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Class>
      <owl:complementOf>
        <owl:Class>
          <owl:intersectionOf rdf:parseType="Collection">
            <owl:Restriction>
              <owl:onProperty>
                <owl:FunctionalProperty rdf:ID="has_first_member"/>
              </owl:onProperty>
              <owl:cardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
              >0</owl:cardinality>
            </owl:Restriction>
            <owl:Restriction>
              <owl:onProperty>
                <owl:FunctionalProperty rdf:ID="has_rest_collection"/>
              </owl:onProperty>
              <owl:minCardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
              >1</owl:minCardinality>
            </owl:Restriction>
          </owl:intersectionOf>
        </owl:Class>
      </owl:complementOf>
    </owl:Class>
  </rdfs:subClassOf>
</owl:Class>
<owl:Class rdf:ID="NonEmptyCollection">
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty>
        <owl:ObjectProperty rdf:ID="has_member"/>
```

```
      </owl:onProperty>
      <owl:minCardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
      >1</owl:minCardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Class rdf:about="#Collection"/>
  </rdfs:subClassOf>
</owl:Class>
<owl:Class rdf:ID="ChoiceOfTerminals">
  <rdfs:subClassOf rdf:resource="#LanguageConstruct"/>
</owl:Class>
<owl:Class rdf:about="#Collection">
  <rdfs:subClassOf rdf:resource="#LanguageConstruct"/>
</owl:Class>
<owl:ObjectProperty rdf:about="#has_member">
  <owl:inverseOf>
    <owl:ObjectProperty rdf:ID="member_of"/>
  </owl:inverseOf>
  <rdfs:subPropertyOf>
    <owl:TransitiveProperty rdf:ID="contains"/>
  </rdfs:subPropertyOf>
  <rdfs:domain rdf:resource="#Collection"/>
  <rdfs:range>
    <owl:Class>
      <owl:unionOf rdf:parseType="Collection">
        <owl:Class rdf:about="#LanguageConstruct"/>
        <owl:Class rdf:about="#LanguageConstruct"/>
      </owl:unionOf>
    </owl:Class>
  </rdfs:range>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="contained_in">
  <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#TransitiveProperty"/>
  <rdfs:domain rdf:resource="#LanguageConstruct"/>
  <rdfs:range rdf:resource="#LanguageConstruct"/>
  <owl:inverseOf>
    <owl:TransitiveProperty rdf:about="#contains"/>
  </owl:inverseOf>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="has_direct_object_part">
  <rdfs:domain rdf:resource="#Aggregate"/>
  <rdfs:range>
    <owl:Class>
      <owl:unionOf rdf:parseType="Collection">
        <owl:Class rdf:about="#LanguageConstruct"/>
        <owl:Class rdf:about="#LanguageConstruct"/>
```

```
        </owl:unionOf>
      </owl:Class>
    </rdfs:range>
    <owl:inverseOf>
      <owl:ObjectProperty rdf:ID="direct_object_part_of"/>
    </owl:inverseOf>
    <rdfs:subPropertyOf>
      <owl:TransitiveProperty rdf:about="#contains"/>
    </rdfs:subPropertyOf>
  </owl:ObjectProperty>
  <owl:ObjectProperty rdf:about="#direct_object_part_of">
    <rdfs:domain rdf:resource="#LanguageConstruct"/>
    <rdfs:subPropertyOf rdf:resource="#contained_in"/>
    <owl:inverseOf rdf:resource="#has_direct_object_part"/>
    <rdfs:range rdf:resource="#Aggregate"/>
  </owl:ObjectProperty>
  <owl:ObjectProperty rdf:about="#member_of">
    <rdfs:subPropertyOf rdf:resource="#contained_in"/>
    <rdfs:domain rdf:resource="#LanguageConstruct"/>
    <owl:inverseOf rdf:resource="#has_member"/>
    <rdfs:range rdf:resource="#Collection"/>
  </owl:ObjectProperty>
  <owl:DatatypeProperty rdf:ID="has_direct_data_part">
      <rdfs:domain rdf:resource="#Aggregate"/>
  </owl:DatatypeProperty>
  <owl:TransitiveProperty rdf:about="#contains">
    <owl:inverseOf rdf:resource="#contained_in"/>
    <rdfs:domain rdf:resource="#LanguageConstruct"/>
    <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#ObjectProperty"/>
    <rdfs:range rdf:resource="#LanguageConstruct"/>
  </owl:TransitiveProperty>
  <owl:FunctionalProperty rdf:about="#has_rest_collection">
    <rdfs:domain rdf:resource="#OrderedCollection"/>
    <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#ObjectProperty"/>
    <rdfs:range>
      <owl:Class>
        <owl:intersectionOf rdf:parseType="Collection">
          <owl:Class rdf:about="#OrderedCollection"/>
          <owl:Restriction>
            <owl:onProperty>
              <owl:FunctionalProperty rdf:about="#has_first_member"/>
            </owl:onProperty>
            <owl:cardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
            >1</owl:cardinality>
          </owl:Restriction>
        </owl:intersectionOf>
      </owl:Class>
```

```
      </rdfs:range>
    </owl:FunctionalProperty>
    <owl:FunctionalProperty rdf:about="#has_first_member">
      <rdfs:domain rdf:resource="#OrderedCollection"/>
      <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#ObjectProperty"/>
      <rdfs:range rdf:resource="#LanguageConstruct"/>
    </owl:FunctionalProperty>
  </rdf:RDF>

  <!-- Created with Protege (with OWL Plugin 2.1 beta, Build 275)  http://protege.stanford.edu
-->
```

B Appendix B: The Upper Ontology for Component Model Constructs

```
<?xml version="1.0"?>
  <rdf:RDF
      xmlns:swrlb="http://www.w3.org/2003/11/swrlb#"
      xmlns="http://www.owl-ontologies.com/componentmodel.owl#"
      xmlns:swrl="http://www.w3.org/2003/11/swrl#"
      xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
      xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
      xmlns:owl="http://www.w3.org/2002/07/owl#"
      xmlns:as="http://www.owl-ontologies.com/abstractsyntax.owl#"
      xmlns:daml="http://www.daml.org/2001/03/daml+oil#"
    xml:base="http://www.owl-ontologies.com/componentmodel.owl">
    <owl:Ontology rdf:about="">
      <owl:imports rdf:resource="http://www.owl-ontologies.com/abstractsyntax.owl"/>
    </owl:Ontology>
    <owl:Class rdf:ID="LanguageConstructHook">
      <rdfs:subClassOf
        rdf:resource="http://www.owl-ontologies.com/abstractsyntax.owl#LanguageConstruct"/>
    </owl:Class>
    <owl:Class rdf:ID="LanguageConstructFragmentBox">
      <rdfs:subClassOf
        rdf:resource="http://www.owl-ontologies.com/abstractsyntax.owl#LanguageConstruct"/>
      <owl:equivalentClass>
        <owl:Restriction>
          <owl:someValuesFrom rdf:resource="#LanguageConstructHook"/>
          <owl:onProperty
        rdf:resource="http://www.owl-ontologies.com/abstractsyntax.owl#contains"/>
        </owl:Restriction>
      </owl:equivalentClass>
    </owl:Class>
    <owl:ObjectProperty rdf:ID="boundWith">
```

```
      <rdfs:range
        rdf:resource="http://www.owl-ontologies.com/abstractsyntax.owl#LanguageConstruct"/>
      <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#FunctionalProperty"/>
      <rdfs:domain rdf:resource="#LanguageConstructHook"/>
    </owl:ObjectProperty>
    <owl:ObjectProperty rdf:ID="extensibleWith">
      <rdfs:range
        rdf:resource="http://www.owl-ontologies.com/abstractsyntax.owl#LanguageConstruct"/>
      <rdfs:domain rdf:resource="#LanguageConstructHook"/>
    </owl:ObjectProperty>
  </rdf:RDF>

  <!-- Created with Protege (with OWL Plugin 2.2 beta, Build 288)  http://protege.stanford.edu
-->
```

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

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

[10] 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/`.

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

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

[13] Uwe Assmann, Jakob Henriksson, Ilie Savga. Prototype component models and composition technology toolset for integration of logic-programming-like REWERSE languages. Technical report, Technical University of Dresden, 2005.