I3-D1

Report on the design of component model and composition technology for the Datalog and Prolog variants of the REWERSE languages

Project number:	IST-2004-506779
Project title:	Reasoning on the Web with Rules and Semantics
Project acronym:	REWERSE
Document type:	D (deliverable)
Nature of document	R (report)
Dissemination level:	PP (restricted to FP6 participants)
Document number:	IST506779/Linköping/I3-D1/D/PP/a1
Responsible editor(s):	I. Savga
Reviewer(s):	M. Kifer
Contributing participants:	Linköping, Dresden, Malta
Contributing workpackages:	I3
Contractual date of delivery:	31 August 2004

Abstract

This design report outlines the elements of a component model for the REWERSE languages. Instead of defining a special component model for each and every language explicitly, we propose a novel concept to derive a component model from a definition of a language, i.e., from its metamodel. The component models will be used together with *invasive software composition* [Aßmann03], and can thus be employed for generic programming, connector-based programming, view-based programming, and aspect-oriented programming. So far, invasive software composition has been developed for Java and has used Java-based component models. In this paper, we discuss the application of this technology for arbitrary languages and ask the question whether it is possible to derive an invasive component model from a given language. To this end, we present two proposals. The first will show that, in principle, every language construct can be generic (*generic construct principle*), that is, it is possible to define an isomorphic mapping from each language construct to the generic elements of its component

model. By generalizing this principle to other languages, we can derive a component model by an isomorphic mapping of the language's metamodel. In our second proposal we will show that it is possible to define semantics for extensibility of components by introducing the notion of list-like language constructs. Thus, it is possible to derive from any given language a component model, which is generic and extensible, and which can thus be used for both generic programming and view-based programming. Once applied to the area of the languages used on the Web, this methodology gives a uniform approach to the composition of ontology languages and Web resources.

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

Report on the design of component model and composition technology for the Datalog and Prolog variants of the REWERSE languages

Ilie Savga[1], Charlie Abela[2] and Uwe Aßmann[3]

[1] Institutionen för datavetenskap, Linköpings Universitetet
Email: ilisa@ida.liu.se

[2] Department of Computing and IT, Junior College, University of Malta
Email: charlie.abela@cs.um.edu.mt

[3] Fakultät Informatik, Institut für Software- und Multimediatechnik, Universität Dresden
Email: uwe.assmann@inf.tu-dresden.de

22 September 2004

Abstract
This design report outlines the elements of a component model for the REWERSE languages. Instead of defining a special component model for each and every language explicitly, we propose a novel concept to derive a component model from a definition of a language, i.e., from its metamodel. The component models will be used together with *invasive software composition* [Aßmann03], and can thus be employed for generic programming, connector-based programming, view-based programming, and aspect-oriented programming. So far, invasive software composition has been developed for Java and has used Java-based component models. In this paper, we discuss the application of this technology for arbitrary languages and ask the question whether it is possible to derive an invasive component model from a given language. To this end, we present two proposals. The first will show that, in principle, every language construct can be generic (*generic construct principle*), that is, it is possible to define an isomorphic mapping from each language construct to the generic elements of its component model. By generalizing this principle to other languages, we can derive a component model by an isomorphic mapping of the language's metamodel. In our second proposal we will show that it is possible to define semantics for extensibility of components by introducing the notion of list-like language constructs. Thus, it is possible to derive from any given language a component model, which is generic and extensible, and which can thus be used for both generic programming and view-based programming. Once applied to the area of the languages used on the Web, this methodology gives a uniform approach to the composition of ontology languages and Web resources.

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

Report on the design of component model and composition technology for the Datalog and Prolog variants of the REWERSE languages

Ilie Savga[1], Charlie Abela[2], Uwe Aßmann[1,3]

1. *Institutionen för datavetenskap, Linköpings Universitetet*
2. *Department of Computing and IT, Junior College, University of Malta*
3. *Fakultät Informatik, Institut für Software- und Multimediatechnik, Universität Dresden*

Abstract

This design report outlines the elements of a component model for the REWERSE languages. Instead of defining a special component model for each and every language explicitly, we propose a novel concept to derive a component model from a definition of a language, i.e., from its metamodel. The component models will be used together with *invasive software composition* [Aßmann03], and can thus be employed for generic programming, connector-based programming, view-based programming, and aspect-oriented programming. So far, invasive software composition has been developed for Java and has used Java-based component models. In this paper, we discuss the application of this technology for arbitrary languages and ask the question whether it is possible to derive an invasive component model from a given language. To this end, we present two proposals. The first will show that, in principle, every language construct can be generic (*generic construct principle*), that is, it is possible to define an isomorphic mapping from each language construct to the generic elements of its component model. By generalizing this principle to other languages, we can derive a component model by an isomorphic mapping of the language's metamodel. In our second proposal we will show that it is possible to define semantics for extensibility of components by introducing the notion of list-like language constructs. Thus, it is possible to derive from any given language a component model, which is generic and extensible, and which can thus be used for both generic programming and view-based programming. Once applied to the area of the languages used on the Web, this methodology gives a uniform approach to the composition of ontology languages and Web resources.

1. Introduction

For the future Semantic Web, reuse will play a major role. Applications will be built from frameworks and components, reusing major parts of applications in the forms of template components, partial component configurations, and product line skeletons. To be able to reuse these partial artifacts, however, component and composition models need to be developed for all involved programming and specification languages [Aßmann03 -PPSWR]. In particular, this holds for logic-based inference languages, since they will play a major role proving the consistency of applications in the Semantic Web [Berners-Lee01]. On the other hand, a major reason why logic languages have not been taken up by the mainstream of software engineering has been that their modularity concepts were rather week. Large applications need a flexible construction out of components, but usually, Prolog or Datalog programs were monolithic and closed, simply, not easy to reuse and to embed into application cores written in other languages. One can go even further and state that: if a logic language does not support both a flexible component model and software composition, then it will be unusable for the Semantic Web.

This dilemma leaves a major challenge for component-based software engineers.

- Will it be possible to develop component and composition models for the rule-based inference languages in the Semantic Web?
- Will it be possible not only to treat OWL, but also other, more powerful languages, as they are envisaged in the Semantic Web layer cake [Berners-Lee01]?

- How quickly can a component model be constructed, if a new deductive language appears?

These questions are the major questions that we, the working group I3 "Composition and Typing" of the Network of Excellence REWERSE [REWERSE], will try to address. The major task is to develop component models for deductive languages. This first report summarizes the results of the group's research, and presents a first attempt to construct a *generic* component model, that is, a component model derived from a given language. As a working case, we will apply our methodology to Prolog, but since it applies to arbitrary languages, it is easy to transfer the reuse technology to other languages of the Semantic Web such as those that will be developed in the REWERSE working group I1.

In the rest of this document, we will try first to come up with an informal and, yet relatively thorough overview on the terminology in the area of component software; then, we will explore the pre- and existing methodologies in the area of software development from the composition point of view; later, we will apply the technique for composing a new dialect language – Prolog. Finally, we propose a novel method of deriving component models from the given source languages and conclude the paper with a discussion on future work.

2. Components and Composition

To lay the basis for our main discussion, in this chapter we introduce some terminology related to the area of component and composition systems.

Why components?

When in the first century BC the Roman Empire reached its prosperity, it grew enormously large. Because of the long distances and bad communication between different parts of the country, the people in power began to have problems in managing the country. In order to alleviate this problem, they decided to break up the country into smaller administrative regions and manage them as much separately as possible. Perhaps, it was the first successful application of the "divide et impera", i.e. "divide and rule" strategy, the technique that has found its application in many areas of social life since then. Among such are politics, sociology and economy; being an important part of our nowadays life, computer science is, of course, not an exception.

Besides the reduced complexity, there is another, even more important result of dividing a software into more or less independent and, still, adaptable parts; this is the "golden dream" of any software developer - reuseability. Programmers are lazy; they do not want to reinvent the wheel each time a slightly new off-the-line product has to be developed. Instead, they would like to possess a methodology that defines how to reintegrate the previously created software into a new context of development, to create software systems from existing software rather than building them from scratch [Kru92].

Not surprisingly, the idea that software should be componentized - built from prefabricated components - appeared in the software community quite a long time ago. In fact, at least one of the definitions of reuse includes the notion of components: "reuse is the use of existing software components in a new context, either elsewhere in the same system or in another system" [Chr94]. At the end of the 60s, this idea was first formulated by Douglas McIlroy [McI68] at the conference in Garmisch, Germany and has had since then a big influence over the computer software.

Components

Usually, when speaking about important concepts, like music, humour, or software agent, a number of, sometimes contradictory, definitions come up to mind; the term "component" is not an exception. Generally, this concept exists in many other disciplines, including mechanical and electrical engineering or architecture; what we are interested in is the notion of a "software component".

Grady Booch believes, that "a reusable software component is a logically cohesive, loosely coupled module that denotes a single abstraction." [Boo87]. Szyperski describes a software component as "a unit of composition with contractually specified interfaces and explicit context dependencies only" that "can be deployed independently and is subject to composition by third parties." He also argues that components have no state and are binary deployables [Szy98]. According to MetaGroup (OpenDoc), "Software components are defined as prefabricated, pre-

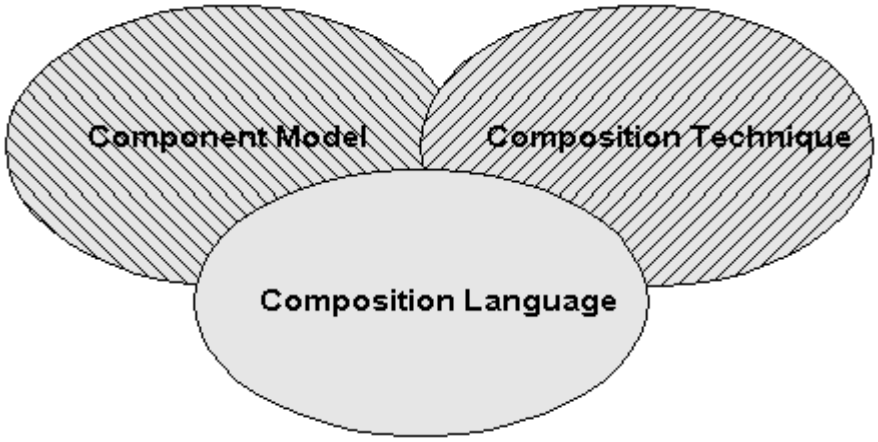

Figure 1. The three main composing parts of a composition system: its component model, the composition language used and the composition techniques applied.

tested, self-contained, reusable software modules - bundles of data and procedures - that perform specific functions."
Intuitively, a component is designed to be composed [Nie95], that is, it is a basic unit for composition. Still, a component is not everything; its structure and behaviour should follow specific rules. Thus, "a software component is a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard."[Hei01]

Component model

What is in this case a component model? In a nutshell, a component model defines how components of a system look. In [Hei01] we find that a component model defines specific interactions and composition standards .

The component model is one of the three core parts of a composition system (see Figure 1). By explicitly describing component interfaces, the model defines the functionality of the components to be used in composition. At the same time, it hides the rest of the components' implementation details, thus providing information hiding. Moreover, it introduces the basis for component substitutability or exchangeability. The latter can be done on the syntactical level, involving typing, and, through semantics, by checking for the components' conformance or contracts.
All of this provides for the *modularity* property of components. Another important and highly desired feature is customizability of components, that is, the ability to parameterize them to different reuse contexts (*adaptability*). Finally, but equally important, the extension of existing components should be easy when new requirements to a system appear (*extensibility*).

Composition technique

Another important part of a component system is the technique used for composition. Basically,this explains the connection between components, their adaptation and extension, the scalability and other aspects of composition. For example, if a composition mismatch occurs (i.e. interfaces of two or more components do not fit witheach other) then an adaptation of some of those composition gates is to be specified (external adaptation). In other cases, gluing code, which consists of a sequence of adaptations, has to be generated to facilitate synchronization, communication and distribution.
In addition, the composition technique also specifies how the integration of components is performed; several examples are

- via base class extension (inheritance),
- via aggregation or
- via integration (mixins).

Moreover, views and aspects can be distinguished; they describe a set of components or a system partially and are added up to the whole. By exchanging them, system functionality can

be varied in complex ways (aspect separation [Kic97]). Finally, the scalability of a composition system should be described, that is, how it can vary in time (static or run-time) and size.

Composition language

The last part of any composition system is its language: this is actually the language that the composition programs are written in. Implicitly or explicitly, it should provide a set of basic, but still expressive enough composition operators (composers), rules that govern the composition as well as integrity constraints to protect the validity of the programs. A composition language can also contain additional features like control flow or protection against run-time exceptions, providing robustness. Another important requirement to such languages is the support of composition abstractions as first-class citizens of the language, that is. a composition language should clearly specify rules and constraints of composition, thus making the architecture of the system explicit. Unfortunately, most of the current systems lack such first-class entities in their grammar; we will discuss this problem in a later chapter.

3. Historical Overview and State of the Art

Software component technology has evolved considerably from modular systems through object-oriented technology up to the current component-based and architecture languages. In fact, if one compares the approaches with the 3 assessment criteria listed above (component model, composition technology and composition language), one gets a uniform picture of the state of the art. In this chapter, we are going to explore this topic, starting with the modular paradigm, then moving on to describe the object-orientation methodology.We then describesome classical component systems, and, finally, we focus on our vision approach; invasive composition.

Modular systems

The primary goal of modules is to split a software product into small and independent subsystems and then to develop and test them separately. Generally, each module hides an important design decision and provides a well-defined interface, which is relatively unlikely to change [Parnas72]. Interestingly, such a packaging was the first attempt to get the exchangeability of software pieces with compatible interfaces and, the most important, to achieve the re-use of already existing modules.

From the composition point of view, the component model of these component systems has been binary code with binding points in form of procedures (import/export) and global variables; the technique implied linking object files statically or at link-time, and, usually, there was no explicit composition language.

The main problem of those languages has been the lack of customizability, in the sence that the pieces of application chopped up by modularization were very hard to customize, should the requirements change slightly. This was primarily due to the lack of parameterization of modules and their very simplified composition mechanism. Still, the module concept has been used inalmost all programming languages, like Modula, C, Ada, Java, C++, Pascal and many others.

Probably, the most interesting example of modular component-based system is *Unix shells (Pipes and Filters)*, proposed by McIlroy (see Figure 2). The component model in this case consists of components with unknown content and binding ports in form of stdin/stdout ports. Components communicate through byte streams with static binding and they can be adapted by filters written in various languages. Finally, there are a number of composition languages, such as shell, C, python and makefiles, which allow for specifying the proper programs, build and version management. Despite, or because, of its simplicity, the UNIX shells style is very flexible and offers one of the most-used component paradigms.

OO systems

To alleviate the inflexibility of modular systems, the object-oriented paradigm was invented. In addition to delegation, it introduced new composition mechanisms, such as inheritance and aggregation, thus allowing for considerably more code reuse than in procedural languages. Components in such systems are either classes (static programming) or objects (at runtime). In the later case, components have identity, persistent state, encapsulated state and behaviour. Moreover, during the binding stage, the receiver of the message is not known, giving additional

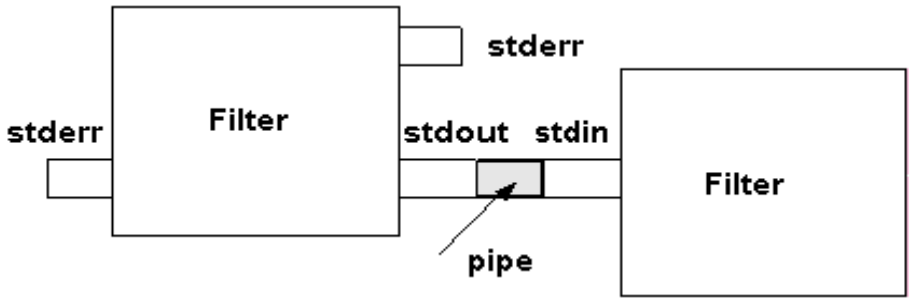

Figure 2. UNIX shells (pipes and filters) architectural style.

power to the polymorphism by this 'late binding' feature. So, the joining points for object-oriented components can be seen as dynamically dispatched polymorphic calls.

Classical component systems

The object-oriented paradigm, with its numerous advantages, has created a fertile soil for the creation of the first commercial component systems. Among the most well known of them are CORBA from OMG [CORBA], JavaBeans [JB] and Enterprise JavaBeans [EJB] from Sun, DCOM [DCOM] and .NET [NET] from Microsoft. Despite of being produced by various companies and appearing differently at first sight, they turn to be rather similar when analysed in more detail.

The component model in such systems usually consist of a set of encapsulated binary components that implement predefined standard interfaces. The basic idea is to completely isolate the implementation and to provide ready-to-use components off-the-shelf (COTS). In addition, the functionality of components is described in terms of binding points using a standard language, for example the *interface definition language (IDL)* [IDL]. Because of this standardized interface description, the actual implementation language of components may vary from Java, C and C++ in CORBA to Microsoft Visual Basic and even Fortran and .NET. Moreover, components have a certain set of predefined properties with standardized names, like having get/set prefixes, which makes the components' introspection easier. As the main objective of such systems is to run as distributed, concurrent systems on various platforms and with permanently evolving software requirements, the composition technique provides means for external adaptation of components by marshalling/unmarshalling and type conversion through the types of standard languages. In addition, some of the systems, like CORBA, support powerful dynamic calls. The composition language also varies from Sun's Java to Microsoft Visual Basic and, generally, suffers from the same problems all other object-oriented languages do when used to describe architecture (we will discuss this issue later on).

Object-oriented languages are not perfect for composition

Unfortunately, despite their numerous advantages object-oriented systems suffer from the lack of expressiveness when defining composition abstractions and rules, since they cannot describe the architecture of the system explicitly. This is mainly due to the fact that object-oriented languages do not support composition abstractions directly as first-class citizens. In fact, such abstractions are usually clearly identified when designing systems and typically recorded as design patterns. Design patterns are "the abstraction from a solution to a recurring problem in a specific context" [GHJV95], meaning that one abstracts from his or her past experience when solving particular type of problems and document this as a design pattern for better reuse in the future. In general, there is no well-known object-oriented language that allows to implement design patterns as first-class entities.These abstractions are scattered throughout the application code, making the software hard to develop, understand, document and maintain.

In order to alleviate these problems, other composition mechanisms were investigated by the software development community , such as Aspect-Oriented Programming [Kic97], Subject Oriented Programming [Har93], Generative Programming [Cza00] and multi-dimensional separation of concerns [Tar99]. The description of these techniques is not dealt here since it is out of scope of this paper.

The Mjolner BETA system

An interesting system we have to mention in this context is the BETA system [BETA]. This system was developed as part of the Mjolner project - a collaborative effort of a number of Scandinavian participants, which aimed for object-oriented software development environments.

Two main parts of the system can be certainly recognized: the proper BETA language and the so-called fragment system [Knu93]. The BETA language is directly designed to enable object-oriented programming. It is a powerful strongly typed language with two main constructs being *pattern* and *object*. Patterns are used to describe classes, procedures, functions and other concepts, while object represents instances of these patterns. In addition, the language is used to implement a series of libraries and application frameworks, which are aimed at automating a number of very common programming problems, like exception handling, file system access or text manipulation. All components in the Mjolner BETA system are constructed from the BETA programming language.

What is interesting in this system in relation to the context of component-based software development, is the adopted solution for modularization, which introduced the notion of *forms*, *slots,* and *fragments*. Formally, a form is a set of non-terminal and terminal symbols derived from a non-terminal according to grammar-specific rules; it is the basic element to define a module in the system. Using specific notation, it is possible to write non-terminals inside forms that can be replaced by other forms during the expansion of these non-terminals. Such elements are called slots since they define openings where other forms may be inserted. Slots have a name and are typed by a meta-class, i.e., an element of BETA's metamodel (the BETA grammar). A BETA slot can be defined by a grammar rule as follows:

Slot ::= '<<' Name':' Metaclass '>>'

where metaclass is an arbitrary element of BETA grammar. Finally, when a form is associated with a name and a syntactic category, it is called a *fragment-form* or simply a *fragment*. So, the fragment system allows for describing fragments as components and their composition interfaces in the form of parameterizable slots.

Table 1 demonstrates a form named Counter of the syntactic category PatternDecl with two slots Up and Down of the same syntactic category DoPart. The form is situated in the file CounterGroup. Now, one can use the slots of this form to bind them with the content of some other form. For example, the code in Table 2 defines a fragment CounterBody that has as its origin CounterGroup. The origin construct specifies that the fragments Up and Down should be substituted for the corresponding slots in CounterGroup.

The result of the substitution - the extent of the fragment - is a combination of the two initial forms and is represented in Table 3. Note, that the parameterization is typed by syntactic category, i.e.metaclass, and, hence, is safe.

Table 1.

```
name ‘home/smith/CounterGroup’
Counter: PatternDecl
Counter:
        (#  Up: (# n: @ Integer enter n <<SLOT: Up:DoPart>> #);
            Down: (# n: @ Integer enter n <<SLOT: Up:DoPart>> exit n #)
         #)
```

Table 2.

```
name ‘home/smith/CounterBody’
origin ‘home/smith/CounterGroup’
Up: DoPart
do n+7 -> n
Down:DoPart
 do n-5->n
```

Table 3.

```
Counter:
        (#  Up: (# n: @ Integer enter n do n+7-> #);
            Down: (# n: @ Integer do n-5->n exit n #)
         #)
```

4. Invasive Software Composition

In this section, we introduce *invasive software composition*, a composition technique generalizing the BETA genericity principle. Beyond *parameterization* of template components, this technology allows for their *extension.* Based on these two basic operations, more powerful operations, such as the management of views or aspects, become possible. For a more indepth discussion about the concepts and applications of invasiveness, refer to [Aßmann03].

Grey-box component model for invasive software composition

In invasive composition, components contain fragments, i.e. pieces of code or XML documents. Fragment components define composition interfaces, which serve as anchors for the composition. Finally, there is a set of composition operators that compose these components by addressing their corresponding anchor points.

From one point of view, this resembles white-box systems, since the components' content is the source code that can be modified by composition operators; on the other hand, the inside of the components is invisibly encapsulated by an interface, like in black-box systems. Such a reuse abstraction is called a *grey-box* and, hence, the composition is *invasive*, as the operators change invasively the fragments in the components [Aßmann03]. To terminologically distinguish such systems from the others, we will call these components *fragment boxes*, the interface anchor points *hooks* and the composition operators *composers*.

In a nutshell, the component model of an invasive composition system is built from fragment boxes that contain hooks, or variation points, - positions of box content, which are subject to change. Generally, there are two types of hooks: *implicit* and *declared*. Implicit hooks are implied by the semantics of the underlying language and, hence, are language-dependent. For example, each Java method has a method entry and a method exit in the same way as each well-formed XML document has a root element - so, the corresponding hooks can be expected. Declared hooks, on the other hand, are explicitly defined by the box writer as parameters in the box's fragments and look like markup tags. This implies extension of the underlying language with new keywords, which follow the standardized naming scheme to be adequately recognized and used for composition. For example, it is possible to add a Java identifier *genericFooIdentifier* to the box code to denote a declared hook named *Foo* of type *genericIdentifier*.

The composition technique then, adapts and extends the components by transforming their hooks, as opposed to the classic black-box transformation (see Figure 3). Adaptation of a hook basically means its fragment-based parameterization with a typed value; it is based on the same

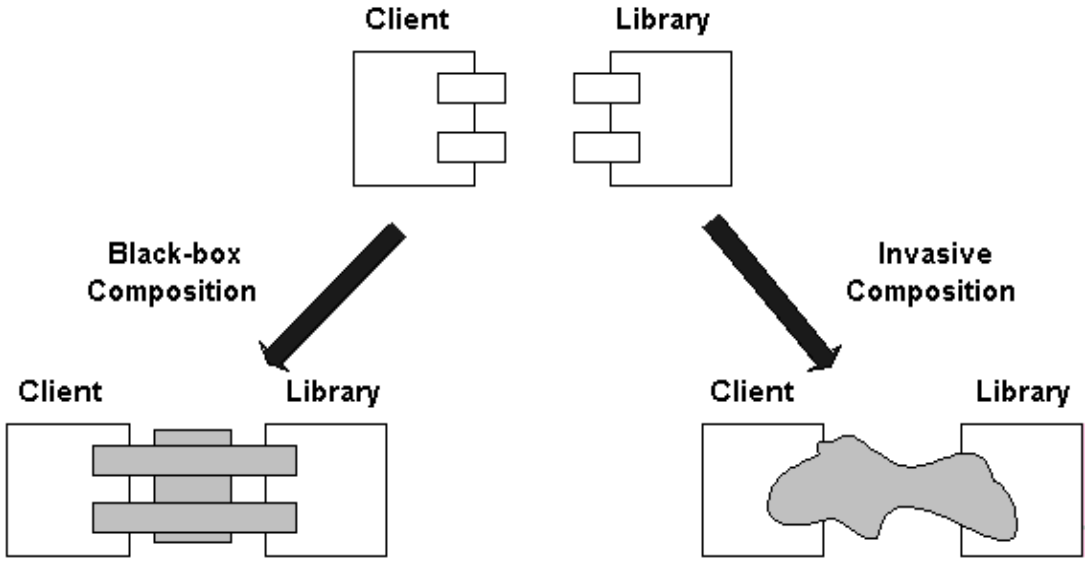

Figure 3. Black-box vs. Gray-box composition. Instead of just generating glue code, composers invasively change the components.

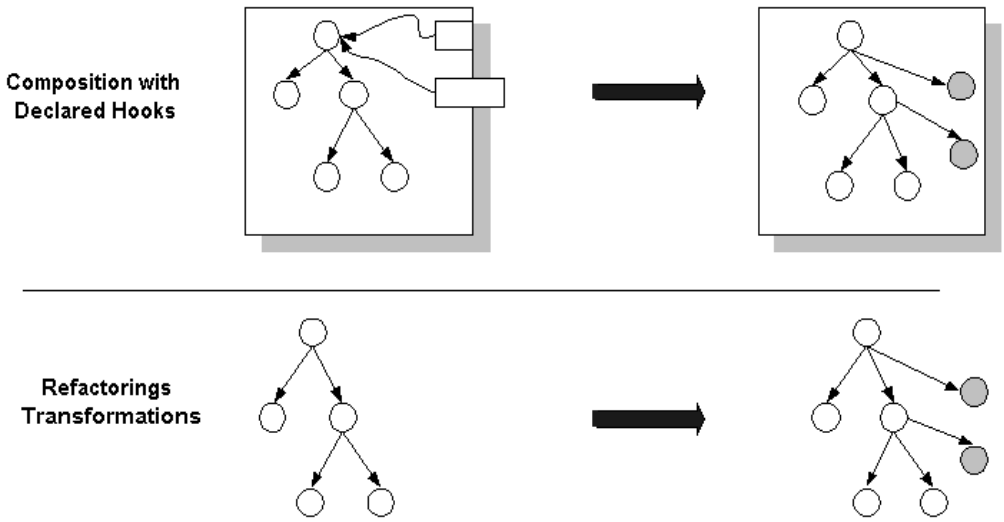

Figure 4. Operations on different levels.

principles as the fragment parameterization in BETA [Knu93] and is performed by the basic operator *Bind*.

A special case of such operation is binding a new name to a component's name hook; the *Rename* operator is used for this name binding. Certain types of hooks can also be extended by making use of the *Extend* operator which takes sets of fragments and enlarges them by additional items. The rest of the operators that form the compositional algebra are essentially based on these first three. For weaving of aspects and intrusive functors binding is used, inheritance and view support rely on extension and connectors combine both techniques with additional glue code generation.

Naturally, the changes resulting from composition on fragment boxes apply directly to the corresponding Abstract Syntax Tree/Abstract Syntax Graph by attaching and removing fragments (see Figure 4). While aspect weaving, view-based and mixin-based composition, and implicit parameterization work on implicit hooks, the template parameterization (see below) and the connectors bind the defined ones. Still, in all the cases the operators work uniformly and do not make any distinction between the types of hooks.

An invasive composition program in such a system can be written in any standard language and represents merely a combination of composition operators from the meta-library. Being very compact comparing to the output code (with ration up to 1/10), such programs enable the description of large systems. Moreover, they are very efficient in terms of the produced code, as only the one that is really needed and described by the composition interface is being generated.

In fact, invasive composition implies staged meta-programming [Aßmann98, Aßmann00]: in the first stage, when the composition program is executed, the whole composition interface in the form of hooks is conceptually made transparentto the boxes and replaced with "real" functional interface. Then, in the second stage, the box fragments are actually evaluated, for example, by a compiler producing machine code. Thus, the composition programs can be thought of as static (comple-time) meta-programs and the discipline of writing them - as staged programming. In order to support meta-programming, that is programming with metaobjects, a system has to support a reflective architecture. Such systems contain programs in the metalevel that reason about the application domain (see Figure 5) or about themselves, modifying their own behaviour [Mae87].

COMPOST - a framework for invasive software composition

In order to provide basic demonstration of the concepts just presented above, we will explore the first existing system for invasive software composition - COMPOST [COMPOST] - by describing shortly some implementation issues and providing several examples of actual

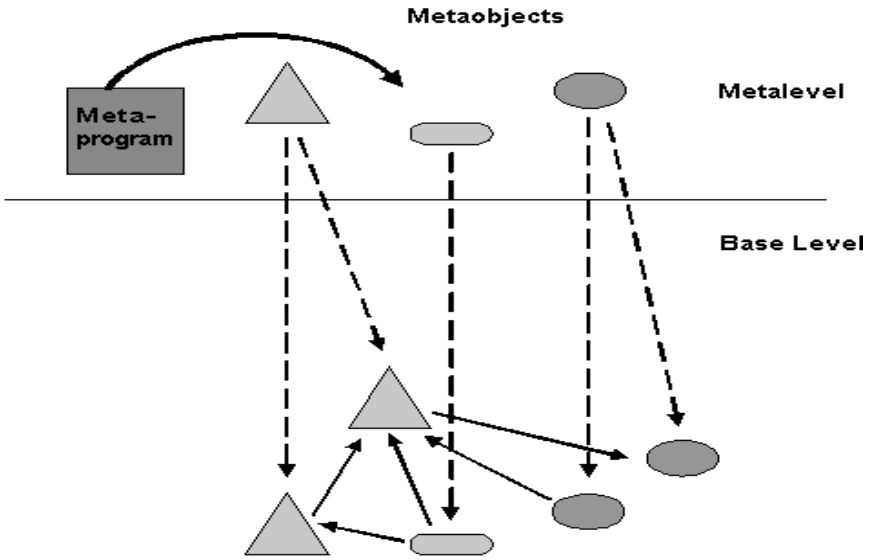

Figure 5. Metalevel architecture: the metaprograms are executed on the metamodel level and are used for computation.

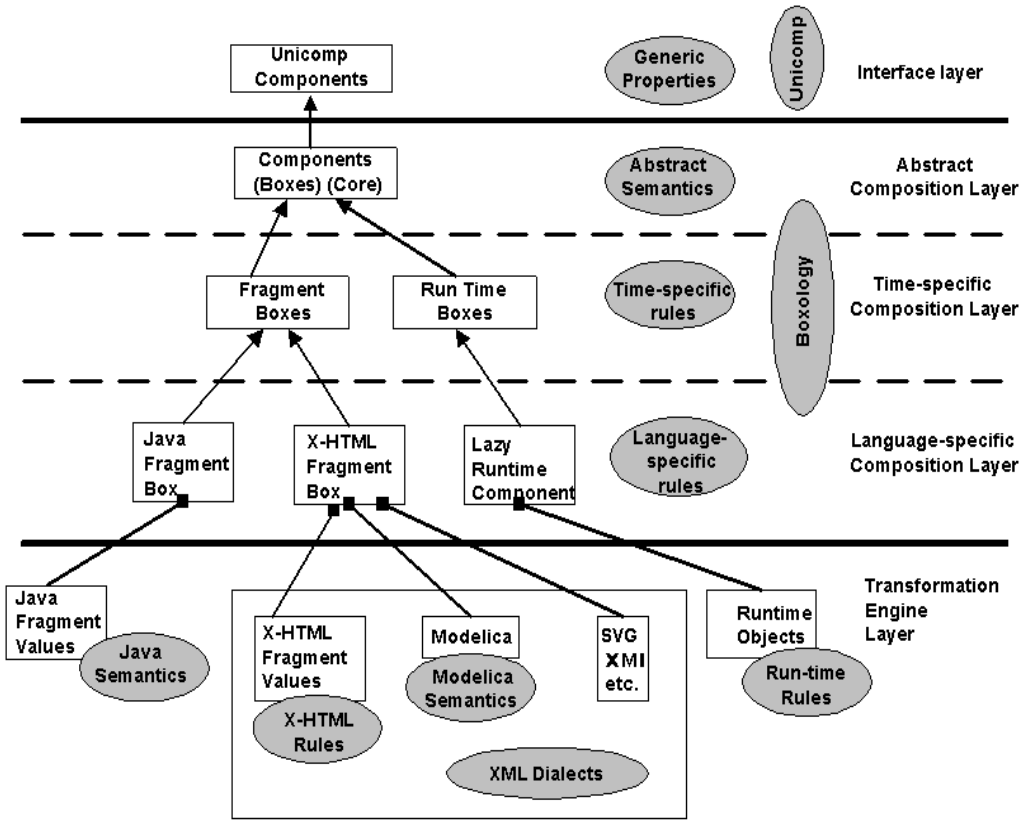

Figure 6. The layered architecture of COMPOST.

The uppermost layer of the system – called Unicomp - contains a set of Java interfaces that abstract away all the fundamental abstractions, thus making the common types for the unified composition. It describes such notions as fragment box, hook, composer, composition program, fragment and composition argument. When working at this level of abstraction, the programmer does not need to know neither the actual component language nor the types of hooks - the composition is done uniformly for all types.
The second, thicker layer is Boxology: as its name implies, it contains the actual implementation of the Unicomp interfaces, as well as the meta-library of composers and some additional supporting features. The implementation goes from the abstract, generic classes down to the concrete, language-dependent ones, for example, Box-> FragmentBox-> JavaBox-> ClassBox or Hook-> FragmentHook-> DeclaredHook-> XMLHook-> XMLElementHook (see Figures 7 and 8).
In fact, one can further divide this layer into three sublayers. The uppermost layer abstracts away the commonalities in the implementation of fragment boxes and hooks for different languages and defines the abstract semantics for all component models in the system. The second sublayer is concerned with temporal constraints that define when the composition takes place: static, i.e. at compile-time, or at run-time. The , language-specific sublayer defines rules specific for each language and language dialect. Currently, the system provides two component models that support two component languages - Java and XML; furthermore, the component model for XML includes extensions for various language dialects, such as XHTML and ModelicaXML [Pop04]. We plan to develop a third model to support Prolog. The issues related to this component model are discussed in the next chapter.
The bottom layer of the system provides the basic program transformation facilities, such as parsing, semantic analysis, cross referencing, actual transformation engine and pretty printing. In addition, there are a number of refactorings that can be applied directly to the language constructs. As currently two languages are supported, the layer includes the packages Recoder [RECODER] for Java and XMLRecoder for XML.
A composition program can be written in Java and looks unsurprisingly like a set of composers from the COMPOST library. All the basic architecture abstractions, required by the grey-box composition, are reified by the system. It is possible to implement mixins, views, aspects, templates and connectors. In addition, the parameterization of all kind of fragments is type-safe

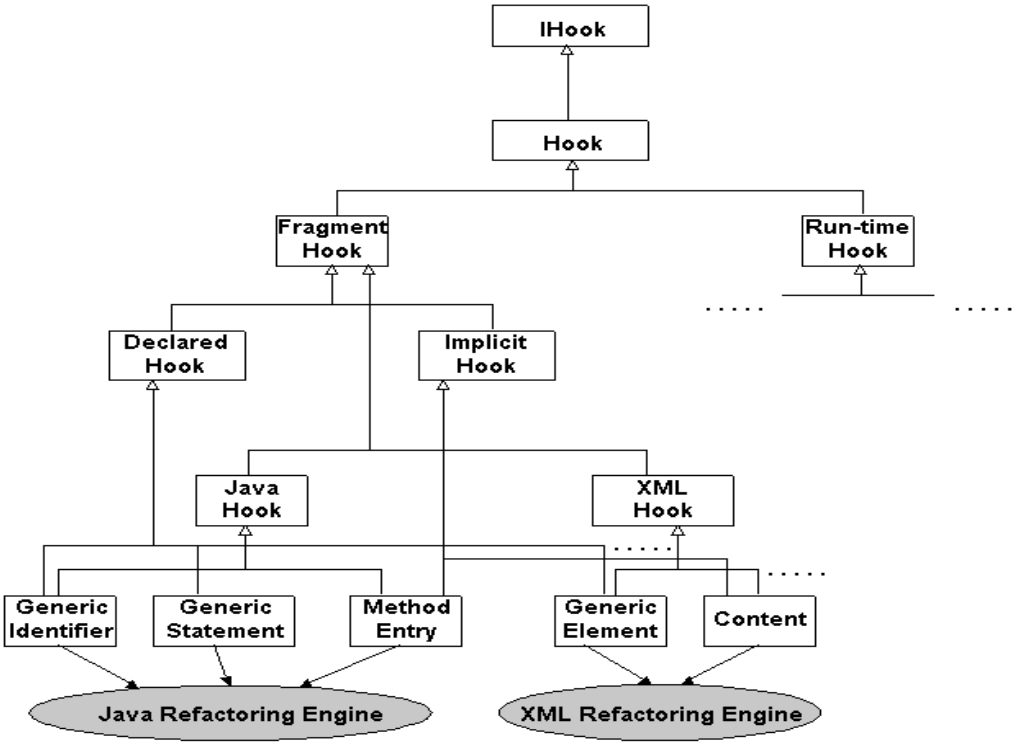

Figure 7. COMPOST Hook Hierarchy**.**

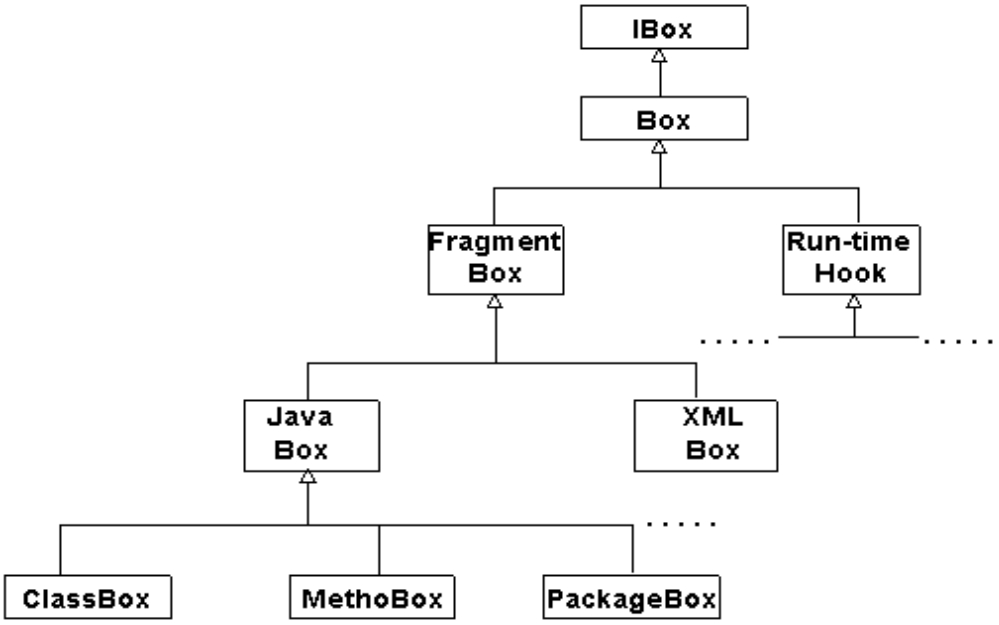

Figure 8. COMPOST Box Hierarchy.

and the adaptation of boxes and glue code is also easily supported. It is also possible to vary strategies of composition in terms of time of execution (eager or lazy), precedence and overwriting of operators.

To demonstrate the implemented composition programs we present several examples, shown below, each consisting of a name, a small description, the actual, rather simplified, program code as well as the initial and resulting component code. All the examples are adapted from [Aßmann03].

Example 1. Weaving debugging aspect.

The composition program adds a parameter and a print statement to the recursive procedure lifeCycle of RecursiveRobot, providing thus the track of the recursion depth. For that, it creates a composition system, finds several implicit hooks and binds them with the corresponding values. Note, that the binding is implicitely typed and, hence, safe. The initial class code and the code after the modification are presented in Listings 1-3.public class RecursiveRobot {

```
  public void lifeCycle() {
    WorkPiece currentPiece = in();
    work(currentPiece);
    out(currentPiece);
    lifeCycle();
}}
```

Listing 1. Recursive robot.

```
public class DemoParameterExtension{
  public static void main(String[] argv) {
    // Prepare the composition by allocating composition system
    CompositionSystem compositionSystem =
                              new FragmentCompositionSystem("gen");
    // Create a fragment box by reading a compilation unit
    Box robot = compositionSystem.createBox("RecursiveRobot.jbx");
    // Add a nesting depth parameter to RecursiveRobot.lifeCycle
    robot.findHook("RecursiveRobot.imports").bind("import boxology.util.Debug;");
    robot.findHook("RecursiveRobot.lifeCycle.parameters").append("int depth");
    robot.findHook("RecursiveRobot.lifeCycle.methodEntry").bind("depth++;
                                          Debug.println(\"Depth \"+depth);");
```

```
    robot.findHook("RecursiveRobot.lifeCycle.methodExit").bind("depth--;");
    compositionSystem.printAll();
}}
```

Listing 2. Composition program extends the recursive procedure with tracing statements.

```
import boxology.util.Debug;
public class RecursiveRobot {
  public void lifeCycle(int depth) {
    depth++; Debug.println("Depth");
    WorkPiece currentPiece = in();
    work(currentPiece);
    out(currentPiece);
    lifeCycle(depth);
    depth--;
}}
```

Listing 3. A new parameter for the debugging version of the recursive robot.

Example 2. Generalized parameterization with generic types and modifiers.
This example shows how the system allows for template programming using explicit hooks in the form of generic types and identifiers. The composition program reads in the template file ArrayList, find the declared hooks of type genericIdentifier, genericSuperclass and genericType and binds them with the corresponding values. This can be done repeatedly, thus producing considerable amount of code. In fact, some parts of COMPOST, like its list hierarchy, were generated using such template instantiation. The result of running ListMaker is presented in Listing 6.

```
public class genericElementIdentifierArrayList extends genericSuperSuperclass
      implements genericElementIdentifierMutableList {
  public genericElementIdentifierArrayList() { super(); }
  public void set(int index, genericTType element){ super.set(index,element); }
  public genericTType getgenericElementIdentifier(int index) {
    return (genericTType)super.getObject(index);}
   public final void add(genericTType element) {  super.add(element); }
}
```

Listing 4. The fragment box for array-based list implementation.

```
public class ListMaker{
  public static void main(String argv[]) {
     FragmentCompositionSystem cs = new FragmentCompositionSystem("gen");
    JavaCompositionSystem compositionSystem = cs.getJavaCompositionSystem
();
    //Read in the names of the list instantiations from a specification file
    //Instead, for the sake of simplicity here we specify the parameters directly
    String type = "Hook", superclass = "Composable", identifier = "Id";
    //Reading list box from file
    CompilationUnitBox box = compositionSystem.createCompilationUnitBox
("ArrayList.jbx");
    //Parameterizing the list box and pretty printing it
    box.findGenericIdentifier("Element").bind(identifier);
    box.findGenericType("T").bind(type);
    box.findGenericSuperclass("Super").bind(superclass);
    box.print();
  }}
```

Listing 5. A composition program for generation of list hierarchies.

```
public class IdArrayList  extends Composable  implements IdMutableList {
   public IdArrayList() {  super();  }
   public void set(int index, Hook element){ super.set(index,element); }
   public Hook getId(int index) {
     return (Hook)super.getObject(index);}
   public final void add(Hook element) {  super.add(element); }
}
```

Listing 6. The result of expanding generic methods to monomorphic instances.

Example 3. Connectors and ports.
As mentioned earlier, the concept of a connector is very important for an explicit architecture description. In COMPOST, we view ports as special instances of a declared hook. As a consequence, connectors can be realized as a special variant of composers. Moreover, we separate other aspects of the architecture, such as topology and transfer, thus generalizing the concept of connectors. The composition program TopologicalConnector reads in two classes - Press and Robot - and binds the out port of the robot to the in port of the press. First, it connects the ports topologically and wires the classes in ProductionCell; the resulting file cannot be compiled yet. Then, it rewrites the topological connection to a concrete one by specifying the actual realization of connection, in this case by buffered local procedure call. Of course, it is possible to connect several classes and obtain a pipe connection, too. The final result of the composition is shown in Listing 9.

```
 public class Robot{
  public void spin(){
    while(true){
      lifeCycle();
     }
   }
  public void lifeCycle(){
    WorkPiece p;
    rotateToTable();
    gate.inWorkPiecePort(p);
    rotateToPress();
    gate.outWorkPiecePort(p);
}}

public class Press{
  public void spin(){
    while(true){
      lifeCycle();
    }
  }
  public void lifeCycle(){
    WorkPiece p = (WorkPiece)gate.inWorkPiecePort();
    close();
    press(p);
    open();
    gate.outWorkPiecePort(p);
}}
```

Listing 7. Robot and Press with COMPOST ports.

```
public class TopologicalConnector{
  public static void main(String argv[]) {
    JavaCompositionSystem compositionSystem = new
              FragmentCompositionSystem().getJavaCompositionSystem();
```

```
      JavaConnectorBox productionCell = compositionSystem.createConnectorBox
("ProductionCell");
      ClassBox press = compositionSystem.createClassBox("Press");
      ClassBox robot = compositionSystem.createClassBox("Robot");
      productionCell.connectTopologically(robot.findOutPort("WorkPiece"),
                                          press.findInPort("WorkPiece"));
      productionCell.produceWireChildren();
      productionCell.connectByBufferedLocalProcedureCall(robot.findOutPort
("WorkPiece"),
                                                         press.findInPort
("WorkPiece"));
      compositionSystem.printAll();
    }}
```

Listing 8. Topological connector in Compost.

```
public class Robot {
  public void spin() {
    while (true) {
      lifeCycle();
    }
  }
  public void lifeCycle() {
    WorkPiece p;
    rotateToTable();
    gate.inWorkPiecePort(p);
    rotateToPress();
    gatePress.putWorkPiece(p);    //fully bound port
  }
  private Press gatePress;
  public void setGate(Press gatePress) {
    this.gatePress = gatePress;
}}

public class Press {
  public void spin() {
    while (true) {
      lifeCycle();
     }
   }
  public void lifeCycle() {
    WorkPiece p = (WorkPiece)getWorkPiece();       //fully bound port
    close();
    press(p);
    open();
    gate.outWorkPiecePort(p);
  }
  private Robot gateRobot;
  public void setGate(Robot gateRobot) {
    this.gateRobot = gateRobot;
  }
  Buffer bufferedWorkPiece;
  public void putWorkPiece(WorkPiece workpiece) {
    bufferedWorkPiece.add(workpiece);
  }
  public WorkPiece getWorkPiece() {
    bufferedWorkPiece.retrieveFirst();
}}
```

```
public class ProductionCell {
  public void wireChildren() {
    Robot robot = new Robot();
    Press press = new Press();
    robot.setGate(press);
    press.setGate(robot);
}}
```

Listing 9. A topological connector generates topologically bound ports.

Example 4. Inheritance as hook extension.
Mixins - classes with anonymous superclasses - are a powerful and flexible abstraction to realize inheritance. In COMPOST, the process of defining mixins is implemented as a composition program operating on implicit hooks. In the following example, the DeviceMaker creates a new class SemiActiveDevice and extends its member list with the members from both PassiveDevice and ActiveDevice, realizing thus multiple inheritance on-the-fly. As opposite to inheritance, it is also possible to use delegation to introduce indirections in the resulting class (not shown here). All the corresponding code is summarized in Listings 10-12.

```
public class ActiveDevice{
  public void spin(){
    while(true) {
      lifeCycle();
     }
  }
  public void lifeCycle(){
    piece = gate.takeIn();
    work(piece);
    gate.pushOut(piece);
}}
public class PassiveDevice{
  private WorkPiece piece;
  public void putPiece(WorkPiece p) { piece = p; }
  public WorkPiece getPiece() { return piece; }
}
```

Listing 10. Passive and Active Devices

```
public class DeviceMaker{
  public static void main(String argv[]) {
    FragmentCompositionSystem cs = new FragmentCompositionSystem("gen");
    Box semiActiveDevice = compositionSystem.createBox("SemiActiveDevice");
    Box passiveDevice = compositionSystem.createBox("PassiveDevice");
    Box activeDevice = compositionSystem.createBox("ActiveDevice");
    semiActiveDevice.extend(activeDevice);
    semiActiveDevice.extend(passiveDevice);
    semiActiveDevice.print();
  }}
```

Listing 11. DeviceMaker creates a new device and mixes in the members from both Passive and Active devices in it.

```
public class SemiActiveDevice {
  public void spin(){
    while(true) {
      lifeCycle();
    }
  }
  public void lifeCycle(){
```

```
    piece = gate.takeIn();
    work(piece);
    gate.pushOut(piece);
  }
  private WorkPiece piece;
  public void putPiece(WorkPiece p) { piece = p; }
  public WorkPiece getPiece() { return piece; }
}
```

Listing 12. The resulting mixin class inherits methods and fields from all of its anonymous parents.

As the presented examples show, the invasive software composition representes a powerful technique that can be used for various purposes. In addition to generic programming and connector-based programming, it can also be employed for view-based programming, multi-dimensional, and aspect-oriented programming. The invasive composition framework under development – COMPOST – supports composition abstractions, e.g. connectors and mixins, as first-class citizens that allow for clear system architecture description. In addition, the framework provides for a rich and extensible library of composion operators, composers, and can be extended to support new component languages.

5. Invasive Software Composition of Prolog Components

For the future Semantic Web, reuse will play a major role. Applications will be built from frameworks and components, reusing major parts of applications in the forms of template components, partial component configurations, and product line skeletons. To be able to reuse these partial artifacts, however, component and composition models need to be developed for all involved programming and specification languages [Aßmann03-PPSWR]. In particular, this holds for logic-based inference languages, since they will play a major role proving the consistency of applications in the Semantic Web [Berners-Lee01]. On the other hand, a major reason why logic languages have not been taken up by the mainstream of software engineering has been that their modularity concepts were rather week. Large applications need a flexible construction out of components, but usually, Prolog or Datalog programs were monolithic and closed, simply, not easy to reuse and to embed into application cores written in other languages. One can even phrase this more provocatively: if a logic language does not support a flexible component model, if the language does not support software composition, it will be unusable for the Semantic Web.

This dilemma leaves a major challenge for component-biased software engineers. Will it be possible to develop component and composition models for the rule-based inference languages in the Semantic Web? Will it be possible not only to treat OWL, but also other, more powerful languages, as they are envisaged in the Semantic Web layer cake [Berners-Lee01]? How quickly can a component model be constructed, if a new deductive language appears? These questions are the major questions of the working group I3 “Composition and Typing” of the Network of Excellence REWERSE [REWERSE]. Its task is to develop component models for deductive languages, and in the following, a first example will be presented, an invasive component model for Prolog.

In the following section we start by giving a very brief description of Prolog and its successor, HiLog, then we discuss their possible component models for these languages and demonstrate how invasive composition can be applied for languages.

Prolog and HiLog

The logic programming language Prolog - PROgramming in LOGic - , was originally designed for natural language processing but has become one of the most widely used languages in different areas of computer science, especially in artificial intelligence. It is based on predicate calculus and enriched with higher-order and meta-level programming. It allows for powerful use of generic predicate definitions, such as sorting and transitive closure. For example, this is how a transitive closure written in Prolog looks:

```
ancestor(X,Y) <- parent(X,Y).
ancestor(X,Y) <- parent(X,Z), ancestor(Z,Y).
```

Still, the language lacks “high-orderness” in its syntax and semantics. For example, only parameter symbols can represent functions, predicates and atomic formulas.
As a solution, a novel language, called HiLog, was invented [Che93]. HiLog has a higher-order syntax, allowing thus for its application in higher-order and modular logic programming, DCG grammars, deductive databases. It supports multiple roles for parameter symbols in a clean and understandable way. More importantly, the language allows arbitrary terms to appear in places where predicates, functions and atomic formulas occur in predicate calculus. For example, this is how transitive closure can be generalized:

```
closure(manager)(X,Y) <- manager(X,Y).
closure(manager)(X,Y) <- manager(X,Z), closure(manager)(Z,Y).
```

Here, the simple predicate ancestor is replaced by an expression depending on the base predicate manager. This illustrates why it is easy to generalize the query to arbitrary predicates: as soon as the predicate manager is replaced by a variable, the query can be instantiated for all base relations.
Although having a higher-order syntax, HiLog, still, preserves its first-order semantics and, hence, soundness and completeness of the proofing procedure. From our point of view, these two languages – Prolog and HiLog - are interesting to be modeled in COMPOST in order to re-apply the existing composition technology for the invasive composition of Prolog components.

The basic idea of invasive composition for Prolog

The basic idea behind the scene is that, despite of its specific "logic" syntax, Prolog programs can be seen as sets of fragments. Thus, once we have Prolog components defined in the same manner as Java and XML ones, we can reuse the existing composition technology to transform them. Moreover, because of this uniformity, the composition becomes transparent and language-independent.
In order for a language to be supported by an invasive software system like COMPOST, its component model must be defined in terms of fragment boxes and hooks. So, we have to provide answers for several basic questions:

- What is a Prolog fragment box?
- What is a Prolog hook?
- Which defined and implicit hooks one can depict in such components?
- Which are possible composition operators that can be applied for transformation?

The extension of the existing component model to support Prolog components involves several steps. First, the fragment box hierarchy is extended; clearly, a Prolog component should subclass FragmentBox in the same manner as the Java and XML components do. Then, Prolog-specific hooks are to be introduced; so far, two of them can easily be imagined:

- The first one, a generic identifier, is similar to the generic identifiers defined for Java and XML components.
- The second one, the members hook, in contrary, is implicit and delimits the content of a Prolog-program, i.e. a set of rules and facts, and resembles the corresponding hook in Java model.

Finally, a Prolog-specific composition system must be defined supporting specific language features, like query evaluation.
Once all these steps are performed, one can start to write composition programs in the same manner and using the same composition library, as for other implemented languages. For example, a generic transitive closure predicate can be defined as follows:

File closureTemplate.pbx

```
genericOutPredicate(X,Y) <- genericInPredicate(X,Y).
genericOutPredicate(X,Y) <- genericInPredicate(X,Z), genericOutPredicate(Z,Y).
```

Here, two declared hooks are declared, namely genericOutPredicate and genericInPredicate, both representing identifiers of type GenericPredicate. Then, one can use such component as a template and supply the corresponding parameters for its instantiation. The composition program ClosureDemo creates a Prolog-specific composition system and a component out of the template and binds its declared hooks. After that, it reads in facts and extends them with the content of the instantiated component. As the main consequence, it is possible to query the resulting component in a Prolog-like manner. The composition program, its input and resulting

components are shown in Listings 13 and 14. This and the following example were inspired by [Che93].

```
public class ClosureDemo{
  public static void main(String argv[]) {
    PrologCompositionSystem cs =  new FragmentCompositionSystem().
getPrologCompositionSystem;
    Box closureBox = cs.createBox("closureTemplate.pbx")
    closureBox.findHook("In").bind("parent");
    closureBox.findHook("Out").bind("ancestor");
    Box factBox = cs.createBox("facts.pbx");
    factBox.extend(closureBox);
    cs.getEvaluator().query(factBox,"ancestor(john,X)").print();
}
```

Listing 13. A composition program binds the explicit hooks of a template fragment box and extends it by a box of facts. The resulting box can be queried and the answers printed out.

File facts.pbx

```
parent(john,bill).
parent(bill,bob).
parent(bob, andrew).
parent(bob, mathew).
```

The content of factBox after its extension and parameterization.

```
parent(john,bill).
parent(bill,bob).
parent(bob, andrew).
parent(bob, mathew).
ancestor(X,Y) <- parent(X,Y).
ancestor(X,Y) <- parent(X,Z), ancestor(Z,Y).
```

List14. The input facts and the result of generic parameterisation.

As mentioned, HiLog allows arbitrary terms to be viewed as functions, predicated and atomic formula. In order to support such features in COMPOST, we extend our closure template with a new hook of type GenericRelation, the subtype of GenericIdentifier.

File closureTemplate2.pbx

```
genericOutPredicate(genericFooRelation)(X,Y) <- genericFoorRelation(X,Y).
genericOutPredicate(genericFooRelation)(X,Y) <- genericFooRelation(X,Z),
                                    genericOutPredicate(genericFooRelation)(Z,Y).
```

The composition program ClosureDemo2 starts with creating of a Prolog composition system. Then, it creates the template component and binds its genericOutPredicate hook. Assume that relationNames is a vector argument, containing the names of some relations, for example, "manager" and "parent". In a loop, the template is repeatedly cloned and instantiated for every such relation. The same is done with the fragment box, containing a specific rule, in this case a rule that determines who should report to whom in an organization. The result of instantiation at each iteration is added to the component factBox, that contains also some facts. The initial fact component and the result of its extension by rules are shown in Listing 15 and 16. Now, it is possible to query the resulting box and obtain, for example, the set of john's ancestors and bosses.

An invasive composition system is not fully higher-order. Since template variables, hooks, are expanded *before* the actual runtime of the component, all second-order features are reduced at compile-time. Hence, using declared hooks, we realize a *static* second-order predicate with a higher-order syntax, but first-order semantics.

```
public class ClosureDemo2{
```

```
    public static void main(String argv[]) {
      PrologCompositionSystem cs =  new FragmentCompositionSystem().
getPrologCompositionSystem;
      Box closureBox = cs.createBox("closureTemplate2.pbx")
      closureBox.findHook("Out").bind("closure");
      Box ruleBox = cs.createBox("reports_to(Person)(Supervisor) <-
                  closure(genericRelationIdentifier)(Person,Supervisor)");
      Box factBox = cs.createBox("facts2.pbx");
      for(int i=0; i<relationNames.size(); i++){
        String name = (String)relationNames.get(i);
        Box closureSnippet = closureBox.copy();
        closureSnippet.findGenericRelation("Foo").bind(name);
        Box ruleSnippet = ruleBox.copy();
        ruleSnippet.findGenericRelation("Foo").bind(name);
        factBox.extend(closureSnippet);
        factBox.extend(ruleSnippet);
        cs.getEvaluator.query(factBox,"reports_to(john)(X)").print();
  }
```

Listing 15. A composition program that realizes generic transitive closure and quieries the resulting box.

File facts2.pbx.

```
parent(john,bill).
parent(bill,bob).
parent(bob, andrew).
parent(bob, mathew).
manager(john,mary).
manager(mary,kathy).
```

The content of factBox after its extension and parameterization.

```
parent(john,bill).
parent(bill,bob).
parent(bob, andrew).
parent(bob, mathew).
manager(john,mary).
manager(mary,kathy).
closure(parent)(X,Y) <- parent(X,Y).
closure(parent)(X,Y) <- parent(X,Z), closure(Z,Y).
reports_to(Person,Supervisor) <- closure(parent)(Person,Supervisor).
closure(manager)(X,Y) <- manager(X,Y).
closure(manager)(X,Y) <- manager(X,Z), closure(Z,Y).
reports_to(Person,Supervisor) <- closure(manager)(Person,Supervisor).
```

Listing 16. The initial facts and the resulting component after parameterization.

In this section, we discussed a possible extension of the current version of COMPOST to support invasive composition of Prolog components. We tried to identify the main issues implied by modeling the Prolog component model as well as possible applications of our approach. For demonstration purposes, we also presented sample composition programs. In the future, we plan to come up with more sophisticated examples and case studies as well as enrichment of the corresponding component model. In the next chapter, we describe future work regarding the automated generation of component models such as the Prolog model and discuss possible solutions for that.

6. Deriving an Invasive Component Model as a Derived Metamodel

So far, both invasive component models have been defined specifically for certain languages, e.g., for Java or for Prolog. In other words, the component models were hand-written. This

section goes an important step forward. It attempts to derive the concepts in the component model automatically from the language definition. We propose a method to derive the concepts in an invasive component model from a given core language, thus automating the process of creating component models. For the Semantic Web, this has the advantage that, whenever a new language should be considered for ontologies, a component model is immediately available, which supports reuse, generic and view-based programming.

The basic idea

The basic idea for the derivation stems from the object-oriented language BETA (see Chapter 3). In BETA, every language construct can be generic, that is achieved by an isomorphic mapping from each language construct to the generic elements of its component model. We claim that by generalizing this principle to other languages, we can derive the metamodel of a generic component model by an isomorphic mapping of the language's metamodel. In addition, we can augment the component model with so-called *list hooks* that form extensible list constructs of a language.

In the following text, we explore this idea, starting with the description of the term *metamodel* and the model hierarchy, then discussing the distinction between two basic composition operators, and, finally, proposing a way to derive a component model from a language's grammar.

Metamodels in the Model Hierarchy

While modeling systems, several levels can be distinguished, forming the model hierarchy [For99, Sch98] (Figure 9). In order to describe them, we use a running example.

Level 0. *The real world.* Consists of the objects in the application domain. For example, describing suite of furniture, we might have a bookcase, a table, a sofa and 6 chairs.

Level 1. *The software world.* Represents the corresponding objects from the real world as software objects. To simulate the furniture, we need to create objects that represent the bookcase, table, sofa and 6 objects to represent the chairs.

Level 2. *The software model.* Abstracts the features and actions of the software objects by means of classes and the relations between objects by relationships. In fact, this level introduces typing for the software objects and their relations. In our furniture example, the chairs can be described with a class Chair. The classes Bookcase, Table and Sofa complete the

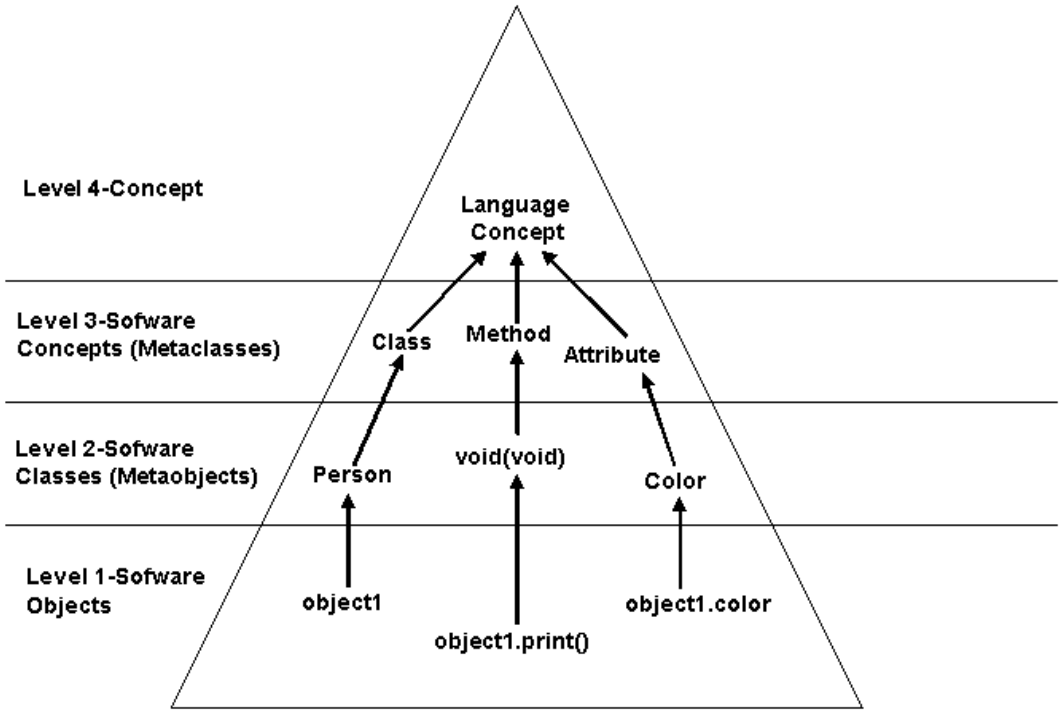

Figure 9. The metalevels of a programming language.

model. As these classes are objects that describe program object from the first level, they are also called *metaobjects*.

Level 3. *The metamodel.* In the same manner as the software model models the first level, the metamodel models the second one. It describes all elements of the model, introducing thus types for the types in the model. As the items on this level describe classes of the level below, they are also called *metaclasses* or *meta-metaobjects*. For instance, the classes of the software model can be described by metaclass Class. Other possible metaclasses could be Property, Method or Attribute items. In essence, the metaclasses of the metamodel define the constructs of a language. Hence, the metamodel is a *language description*.

Also, component models live on this level: they describe reuse-oriented typing schemes for the programs on level 2. Their concepts (e.g., fragment boxes, hooks, or composers) form metaclasses of a specific metamodel, i.e., the component model.

Level 4. *The metametamodel.* This is the most abstract layer of the hierarchy that describes all the items of the metamodel. In fact, it describes the concepts that can be used to specify a programming language. In other words, the third level introduces programming languages and the fourth level defines the concepts for their specification. The metametamodel forms a language specification language. Metamodels are *written in* a metametamodel. Examples for metametamodels are

- grammar formalisms such as EBNF [EBNF] (in which metamodels, grammars, can be written),
- attribute grammar formalism, such as ELI-LIGA (in which metamodels, attribute grammars, can be written) [ELI],
- the metaobject facility (MOF) of OMG [MOF], a UML-class-diagram-like specification language, which can describe class-diagram-based metamodels, e.g., for UML. Another application of the MOF is *type-system mapping*. The concepts defined in MOF can be mapped to the types of a specific type system and from these mappings it is possible to generate code that allows for navigating in object graphs. Thus, services to operate on types systems are automatically derived and, moreover, different type systems can be compared under the same unifying model.

Our core idea is to use the metamodelling approach for describing component models of Semantic Web languages.

The Bind and Extend Composition Operators

The two basic operators of the composition calculus – Bind and Extend – were implicitly introduced when describing invasive composition; we shortly investigate them before going into details of our proposal. In fact, they complement each other and yield a strong composition operator algebra, since all other operators can be produced using these two basic ones [Aßmann2003].

Which prerequisites do both operators pose on the underlying languages? Bind is used for (typed) parameterization of generic slots and requires *genericity* and *adressibility*. Genericity means that it can be specified for a language construct that it is generic, i.e., that a component parser can recognize a generic slot easily. Secondly, the slot should be named, so that it can be addressed from a composition operator (*adressibility).*

On the other hand, some constructs in a language can be thought of as list-like and, hence, can be extended. For example, the members of a Java class or the rule list of a Prolog file can be treated as lists and extended by applying the composition operator Extend. Since the Extend operator can be repeatedly applied, it allows for view-based programming, thus repeatedly extending of components depending on the context of their use [Aßmann2003]. Since view-based system development directly generalizes inheritance, inheritance can also be realized as hook extension.

A Component Model as a Derived Metamodel

As previously mentioned, the concepts of a language are specified by its metamodel, which is on the third level of the model hierarchy (see Figure 9 in Chapter 6). Once they are defined, it is possible to manipulate them using the facilities from the meta-metamodel, like the MOF does. Our main idea is to generalize the concept of the generic language construct by defining

a mapping from each entity in the language's metamodel to the elements of the component model. In addition, for extensible component hooks, we propose to define extension points for every list construct in a language. Both the generic model and the extension model form the full component model. We call the language the *core language.* A component model, which is defined by a mapping from a metamodel is called a *derived component model*. Since the derived component model is used to specify composition interfaces of components, the *augmented language* is the core language joined with the composition interface language.

1. *Mapping for genericity***.**

Suppose, we have a core language without generic constructs. If we want to define generic constructs for all language constructs, there should be an isomorphic mapping from the elements of the metamodel to a second metamodel, the metamodel of the generic constructs.

genGenericModel:Metamodel -> Metamodel

That is a mapping that maps a language metamodel to an isomorphic metamodel of generic constructs. If the metamodel is a graph, the generic construct metamodel is also a graph.

In the case of BETA, the generic slot concepts can be constructed by mapping every language construct to a generic slot concept. Hence, the generic slots result by systematically applying a mapping from the language constructs to the elements of the composition interfaces. The resulting model is generic and the basic operator that works on its components is Bind.

2. *Mapping for extensibility.*

Not all language elements, however, can be extended. Here, the idea is to define a partial component model. Let

genExtensibleModel:Metamodel ->Metamodel

be a unary composition operator, which derives from all lists in the core language's metamodel. For every list-like fragment of a language, we can, atleast derive a list-like extensible fragment that marks up parts of a list.

ExtensibleFragment ::= [[<name> : <metaclass>]]

The Extend operator can be applied to this fragment. It will extend the extensible construct with a fragment, typed by a metaclass.

The extensible constructs can be regarded as elements of the composition interfaces of components. Hence, the elements form a small language for the extensible part of the composition interfaces of a component. Again, the composition interface language can augment the core language, such that components can be extended very easily.

7. Deriving a Prolog Component Model: an Example

In this section we combine the previously mentioned ideas with the ideas of Chapters 4 and 5 to derive an invasive component model for Prolog. The model generalizes HiLog in two ways: first, by allowing all language elements to be generic, and then all list-like language elements to be extensible. Finally, and more important, the component model is derived systematically and not ad-hoc.

We make use of the following, simplified grammar for a Prolog-like language:

```
PrologGrammar = (N={Language, Rule, Head, Body, Clause, TermOrVar, Var, Term,
Functor, SmallWord, Predicate},
    T={ ',', ':-', '.', SmallWord}
    Z=Language,
    R={Language ::= Rule * .                              -- (1)
        Rule ::= Head ':-' Body .
        Head := Clause .
        Body ::= Clause // ',' '.'.                       -- (2)
        Clause ::= Predicate '(' TermOrVar // ',' ')' .-- (3)
        TermOrVar ::= Var .
        Var ::= LargeWord
        Term ::= Functor '(' TermOrVar // ',' ')' .     -- (4)
        Functor ::= SmallWord.
        Predicate ::= SmallWord.
    },
```

).

Using a grammar is not the only way to specify the structure of a language. In a UML model, knowing the structure allows for other part-of relationships to be modelled. Hence, a UML model can also express static contextual relationships, that is, context conditions. Similarly, a model in OWL would be possible. However, for this example, we will use the grammar formalism. We assume it has the meta-metamodel

```
Grammar = ( Nonterminals, Terminals, Rules, StartNonterminal )
```

Genericity

Using the basic idea for a derived generic component from above, the nonterminals can be made to be generic slots:

```
genGenericModel: Nonterminals --> Nonterminals
-- every nonterminal is mapped to a generic counterpart
genGenericModel(N) := { G<A> | where A in N }
-- we use the < > string concatenation operator as in C++

genGenericModel(PrologGrammar.N) = {GLanguage, GRule,GHead, GBody,
GClause, GTermOrVar, GVar, GTerm GFunctor, GPredicate}
```

where the nonterminals starting with "G" are generic nonterminals, which can be used as placeholders for a sentential form of their concrete counterparts.
Additionally, a new rule set must be derived, whose heads are the new nonterminals that express genericity:

```
genGenericModel: Rules --> Rules
genGenericModel(R) := { NewHead ::= "<<" Name ":" Head ">>".
                              | r in R, NewHead = genGenericNonTerminals (r.Head) }
```

from which results for our example:

```
genGenericModel (PrologGrammar.R) = {
        GLanguage ::= "<<" Name ':' Language ">>".
        GRule ::=  "<<" Name ':' "Rule" ">>".
        GHead := "<<" Name ':' "Head" ">>".
        GBody ::= "<<" Name ':' "Body" ">>".
        GClause ::= "<<" Name ':' "Clause" ">>"
        GTermOrVar ::= "<<" Name ':' "TermOrVar" ">>" .
        GVar ::= "<<" Name ':' "Var" ">>".
        GTerm ::= "<<" Name ':' "Term" ">>".
        GFunctor ::= ."<<" Name ':' "Functor" ">>".
        GPredicate ::= "<<" Name ':' "Predicate" ">>".
}
```

The Prolog grammar would then be extended by pointwise set union as follows:

```
PrologGrammarWithGenerics = (
    Nonterminals + GenericNonterminals,
    Terminals + { "<<", ":", ">>" }
     Z=NLanguage,
     PrologGrammar + {
         Nlanguage ::= Language | GLanguage.
         Nrule ::= Rule | GRule.
         Nhead ::= Head | GHead.
```

```
        ...
        Npredicate ::= Predicate | GPredicate.
})
```

In this grammar, invasive components, i.e., template components can be specified, for instance the generic transitive closure from Chapter 5:

```
component c = {
   <<TransRelation: Predicate>>(X,Y) :- <<BaseRelation:Predicate>>(X,Y).
   <<TransRelation:Predicate>>(X,Y) :- <<TransRelation:Predicate>>(X,Z),
                                       <<BaseRelation:Predicate>>(Z,Y).
}
```

The Bind, composition operator for generic components, expands a slot from such a template into a concrete sentential form:

```
Bind : Component x Slot x Fragment --> Component
Bind(A, A.Name == "<< Name ":" Type ">>", value) :=  A.Name = value
```

Since for all nonterminals generic slots are available, the language is fully generic. The composition interface of a fragment component consists of all used generic nonterminals.

Extensibility

Similarly for the genGeneric mapping, we can also define a mapping that derives extensible language constructs from the constructs of a language. Thereby, it defines an extensible component model for the language. First, we define its behaviour on nonterminals:

```
genExtensibleModel: Nonterminals -> Nonterminals
genExtensibleModel(N) := { E<B>  | B in listHeads(N),
  listElements(N) =  { ListElement | r in N
        and r == Head ::= ListElement //
        or   r == Head ::= ListElement *
  }
```

i.e., listElements selects all nonterminals that characterize elements in list-like constructs, and these generate the extensible nonterminals. For our grammar,

```
listElements(PrologGrammar.N) = { Rule, Clause, TermOrVar}
```

holds, which is determined by rules (1-4). Hence, for the grammar the nonterminals

```
   N-Ext = genExtensibleModel(PrologGrammar.N)
         = { ERule, EClause, ETermOrVar}
```

results. Then, we give another function that produces terminals for them in the concrete language:

```
genExtensibleModel(R) ::= { NewHead ::= "[[" Name ":" Head "]]".
                            | r in R, NewHead = genExtensibleNonterminals(r.Head) }
```

from which results for our example:

```
genExtensibleModel(PrologGrammar.R) = {
        ERule ::=  "<<" Name ':' "Rule" ">>".
        EClause ::= "<<" Name ':' "Clause" ">>"
        ETermOrVar ::= "<<" Name ':' "TermOrVar" ">>" .
```

```
}
```

The Prolog grammar would then be extended by pointwise set union as follows:

```
PrologGrammarWithExtensibility = (
    Nonterminals + ExtensibleNonterminals,
    Terminals + { "[[", ":", "]]" }
     Z=NLanguage,
     PrologGrammar + {
          Nlanguage ::= Language | ELanguage.
          Nrule ::= Rule | ERule.
          Nhead ::= Head | EHead.
          ...
          Npredicate ::= Predicate | EPredicate.
     }
)
```

In this grammar, we can write invasive components that almost look like templates, but are extensible. For instance, in the transitive closure example we can introduce a extensible hook that can be extended with the Extend operator:

```
component c = {
   base(X,Y,[[baseExt:TermOrVar]]) <- base(X,Y,[[baseExt:TermOrVar]]).
   trans(X,Y)                      <- trans(X,Z), base(Z,Y,[[baseExt:TermOrVar]]).
}
```

and define an extension operator extended

```
Extend: hook, value ->
Extend : Component x Hook x Fragment --> Component
Extend(A, A.Hook == "[[" Name ":" Type "]]", value) :=
         A.Hook = [[ A.Hook, value ]]
```

where [[ ]] means insertion into embedding lists. Then, we can operate

```
Extend(c, c.baseExt, Attribute) = {
   base(X,Y,Attribute, [[baseExt:TermOrVar]]) :-
        base(X,Y,Attribute, [[baseExt:TermOrVar]]).
   trans(X,Y) :- trans(X,Z), base(Z,Y,Attribute, [[baseExt:TermOrVar]]).
}
```

and

```
Extend(c, c.baseExt, []) = {
base(X,Y) :- base(X,Y).
trans(X,Y) :- trans(X,Z), base(Z,Y).
}
```

Since for all list-contained nonterminals hooks are available, the language is fully extensible for all its list-like constructs. The composition interface of a fragment component consists of all used extensible nonterminals.

Both genericity and extensibility

The full component model is both generic and extensible, because the above definitions are in fact a union in the model.

```
PrologGrammarWithGenericsAndExtensibility = (
    Nonterminals + GenericNonterminals + ExtensibleNonterminals,
    Terminals + { "<<", ">>", "[[", ":", "]]" }
     Z=NLanguage,
     PrologGrammar + {
          NLanguage ::= Language | GLanguage | ELanguage.
          NRule ::= Rule | GRule | ERule.
          NHead ::= Head | GRule | EHead.
          ...
          NPredicate ::= Predicate | GPredicate | EPredicate.
     }
)
```

The composition interfaces of components in the core language under this derived component model consist of all used generic slots and extensible hooks. Components become possible such as:

```
component c = {
    <<TransRelation: Predicate>>(X,Y, [[transExt1:TermOrVar]]) :-
         <<BaseRelation:Predicate>>(X,Y, [[baseExt1:TermOrVar]]).
    <<TransRelation:Predicate>>(X,Y, [[transExt2:TermOrVar]]) :-
          <<TransRelation:Predicate>>(X,Z, [[transExt3:TermOrVar]]),
          <<BaseRelation:Predicate>>(Z,Y, [[baseExt2:TermOrVar]]),
          [[clauseExt:Clause]].
    [[ruleExt:Rule]]
}
```

This component is parameterized and extended at several points. An imperative composition programming

```
Bind(c, c.transRelation, grandchild);
Bind(c, c.baseRelation, child);
Extend(c, c.transExt1, 0);
Extend(c, c.transExt2, RecursionDepth+1);
Extend(c, c.transExt3, RecursionDepth);
Extend(c, c.clauseExt3, print("found another grandchild", Y));

Extend(c, c.ruleExt,[]);
Extend(c, c.clauseExt,[]);
Extend(c, c.transExt1, []);
Extend(c, c.transExt2, []);
Extend(c, c.transExt3, []);
Extend(c, c.baseExt1, []);
Extend(c, c.baseExt2, []);
```

will result in a specific form of transitive closure, with recursion depth tracking:

```
c == {
grandchild(X,Y,0) :-
      child(X,Y).
grandchild(X,Y,RecursionDepth+1) :-
      grandchild(X,Z,RecursionDepth), child(Z,Y),
print("found another grandchild", Y)
}
```

8. Discussion

The reader has certainly discovered that a second-order language, here a second-order logic programming language directly provides the component models, in the form of higher order functions and clauses. However, full second-order logic programming is undecidable. Hence, one could say that the decisive difference of a derived component model to a second-order language is that the invasive composition operations are executed in a stage *before* the actual system (*staged metaprogramming*) [Tah97]. Hence, a derived component model in our sense is nothing else but an application of staged metaprogramming principles.

The advantage of a derived component model is that it is automatic. It comes for free. Variation and extension points of components are systematically named, and can be addressed from outside the components during the composition. Whenever a language is defined, the designer immediately knows how to use extension and variation points, because the roles of the nonterminals in the grammar determine them.

In MDA, various horizontal transformations can be defined [MDAGuide]. A horizontal transformation does not change the level of abstraction, is applied to the concepts of the same language, but refactors the model. This is not quite what we have been doing - we have been extending our modelling language. Hence, we could say that a derived component model is a horizontal extension, in the sense, that the modelling language is extended with concepts for reuse.

Lämmel's work has applied static metaprogramming to attribute grammars [Lämmel98]. He uses the Lambda calculus as a composition language (with the Bind operation), simulates Extend operations by function composition, and is able to compose attribute grammar components. Our work is in this spirit; however, we have made the Extend operation explicit, which allows us for deriving an extensible component model. This is not possible in Lämmel's approach.

Extended derived component model.

Up to this point, we discussed *automatic component models,* i.e., the models were generated from the grammar of core languages. Since the generation treats only the syntactical aspects of a language, we call them syntactical models. Such a model is always isomorphic to a proper subset of the core language and, hence, is homomorphic to the language. Nevertheless, a language designer might want to add additional semantics to the model in the form of variation and extension points. For example, to model component ports one would like to extend a model with the component-port concept. Clearly, this concept introduces additional semantics, since for a composition operation semantic conditions have to be taken into account. We call such extensions semantic variation points. Note, that such component models are no longer homomorphic to the metamodel of the core language.

Restricted derived component models.

Alternatively, a concrete component model can select a subset of the available slots and hooks in the derived syntactic model to constrain the composability of the components. Then, not all language constructs are variation or extension points, but the composition is restricted to a subset of them. Again, the mapping function in this case is not monomorphic, but epimorphic.

Restricted and extended derived component models.

Probably, the most realistic scenario is a derived model for a restricted subset of the language, enriched by additional semantic constructs. The restriction in such cases is applied because the language designer is not interested in all the language constructs to be used in composition interfaces. Onthe other hand, the semantic extension is due to the specific requirements for the composition, which are not covered by the syntactical part of a model.

There are also other open questions. Is it possible, beyond the generic and extension models, to derive further dependent metamodels for reuse? In other words, we got now some automatic reuse mechanisms with a language, can we get more? How do frameworks with the derived models look like? The markup tags for parameterization and extension have to be explicitly specified. What about implicit hooks that allow for unforeseen extensions?

In this and the previous chapters we discussed our vision of the consistent construction of a generalized component model for various component languages. We have shown that there exists a simple form of invasive component models that come for free. We believe that, used in a systematic way, this technology can be applied to a range of languages, such as Prolog, Java,

XML, or OWL, and will help to achieve a unified and useful reuse technique for modelling in the Semantic Web and beyond.

9. Conclusion

In the first part of the paper we gave a short survey of the related work. One of the future tasks will be to investigate the commonalities in different methodologies, and try to come up with unifying concepts. For example, there are interesting common points in generative programming and software composition or ADL and invasive composition that, once investigated and formulated, can contribute to all techniques.

We have also given an overview of software composition, presented invasive software composition for Java, and also suggested a way how to derive an invasive component model for any given language from the language's metamodel. A component model for Prolog was described, and this extends HiLog for extension-based composition.

As regards the Prolog modeling, we are just starting to investigate the issues involved. The semantics of the language will be investigated in more depth and the corresponding component model thoroughly described. After that, we also plan to extend the model for other closely related languages, such as Datalog, and OWL. Also, serious efforts will to be made to transparently unify all the models within the framework.

Still, our main future scope is to elaborate a novel technique for deriving important parts of the concepts in an invasive component model from a given core language. Generalizing the generic language construct principle and applying it to a number of various languages, we aim to achieve the metamodel of a generic component model for the given languages, and, thus, get a uniform framework of fragment component transformation. In this context, the main future work will consist in formalizing our approach and implementing software for the tool support.

References:

[Ach02] Acherman, Franz. *Forms, agents and channels.* PhD.thesis. University of Bern, 2002

[Ach01] F. Achermann, M. Lumpe, J.-G. Schneider, and O. Nierstrasz. *Piccola - a small composition language*. In H. Bowman and J. Derrick, editors, Formal Methods for Distributed Processing, an Object Oriented Approach. Cambridge University Press, 2001

[Ach01+] F. Achermann and O. Nierstrasz. *Applications = Components + Scripts - A Tour of Piccola*. In M. Aksit, editor, Software Architectures and Component Technology. Kluwer, 2001.

[Aßmann98] Aßmann, Uwe. *Meta-programming composers in second-generation component systems.* In Bishop J. and Horspool N., editors, System Implementation 2000 – Working Conference IFIP WG 2.4, Berlin. Chapman and Hall.

[Aßmann00] Aßmann, U., Genssler, T., and Bär, H. *Metaprogramming Grey-Box Connectors.* In Mitchell, R., editor, Proceedings of the International Conference on Object-Oriented Languages and Systems (TOOLS Europe). IEEE Press, Piscataway, NJ.

[Aßmann03] Aßmann, Uwe. *Invasive software composition.* Springer-Verlag, 2003.

[Aßmann03-PPSWR] *Composing Frameworks and Components for Families of Semantic Web Applications.*

[BETA] The BETA language homepage. http://www.daimi.au.dk/~beta/

[Bra90] G. Bracha and W. Cook. *Mixin-based inheritance.* In Proceedings OOPSLA/ECOOP'90, ACM SIGPLAN Notices, pages 303-311, Oct. 1990.

[Bau97] Baumer D., Gryczan G., Knoll R., Lilienthal C., Riehle D. and Zullighoven H.. *Framework Development for Large Systems*. In Communications of the ACM, Volume 40, Number 10 (October 1997), pages 52-59, October 1997.

[Berners-Lee01] Tim Berners-Lee, James Hendler and Ora Lassila. *The Semantic Web.* Scientific American, May, 2001.

[Boo87] Booch, Grady. *Software Component with ADA.* 1st edition Software Component with ADA, 1st edition. Benjamin-Cummings Publishing Co., Inc, 1987

[Chr94] B. Christine and Marciniak John J. *Encyclopedia of software engineering*, 1994.

[CORBA] *Catalog of CORBAr/IIOPr specifications*
www.omg.org/technology/documents/corba_spec_catalog.htm

[Che93] W. Chen, M. Kifer, D.S.Warren. *HiLog: a Foundation for Higher-Order Logic Programming.* 1993

[COMPOST] *COMPOST homepage*. www.the-compost-system.org

[Cza00] K. Czarnecki and U. Eisenecker. *Generative Programming. Methods, Tools, and Applications.* Addison-Wesley, 2000.

[DCOM] *DCOM homepage*. www.microsoft.com/com/tech/DCOM.asp

[For99] Forman, I.R. and Danforth, S.H. Putting *Metaclasses to Work: a New Dimension in Object-Oriented Programming.* Addison-Wesley Longman, Redwood City, CA. 1999.

[Kic97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. *Aspect-oriented programming.* In Proceedings of ECOOP'97, pages 220-242. Springer Verlag, 1997. LNCS 1241.

[EBNF] *Extended BNF*. ISO/IEC standard.
http://www.iso.org/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=26153

[EJB] *Enterprise JavaBeans specification.* http://java.sun.com/products/ejb.docs.html

[ELI] ELI project homepage. http://**eli**-project.sourceforge.net

[Fre83] P. Freeman. *Reusable software engineering: Concepts and research directions.* 1983.

[Har93] W. Harrison and H. Ossher. *Subject-oriented programming (A critique of pure objects)*. In A. Paepcke, editor, OOPSLA 1993 Conference Proceedings, volume 28 of ACM SIGPLAN Notices, pages 411-428. ACM Press, Oct. 1993.

[Hei01] G. T. Heineman and W. T. Councill, editors. *Component-Based Software Engineering.* Addison Wesley, 2001.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns.* Addisson-Wesley, 1994.

[IDL] *The OMG IDL ISO International Standard*. www.iso.ch/cate/d25486.html

[JB] *JavaBeans specification.* http://java.sun.com/products/javabeans/docs/spec.html
[Java] *Java 1.4.2 API specification.* http://java.sun.com/j2se/1.4.2/docs/api/
[Joh88] R. Johnson and B. Foote. *Designing reusable classes.* Journal of Object-Oriented Programming, 1(2):22-35, June 1988.
[Knu93] J. Lindskov Knudsen, M.Lofgren, O.Lehrmann Madsen, B.Magnusson. *Object-oriented environments: the Mjolner approach.* Prentice Hall, 1993
[Lämmel98] Lämmel, Ralf. ???
[Mae87] Maes, P. *Concepts and experiments in computational reflection*. In Proceedings of OOPSLA87, number 12(22) in ACM SIGPLAN Notices, pages 147-155. New York, 1987.
[McI68] McIlroy, Douglas. *Mass-produced software components.* In P. Naur and B. Randell, editors, Software Engineering, NATO Science Committe report, pages 138--155, 1968.
[Mil] Milner, Robin. *A polyadic pi-calculus: a tutorial.* citeseer.ist.psu.edu/milner91polyadic.html
[MOF] *The OMG MOF specification*. http://www.omg.org/cgi-bin/doc?formal/00-04-03
[.NET] *.NET homepage.* http://www.microsoft.com/services/net/default.asp
[Nie95] O. Nierstrasz and L. Dami. *Component-oriented software technology.* In O. Nierstrasz and D. Tsichritzis, editors, Object-Oriented Software Composition, pages 3-28. Prentice-Hall, 1995.
[Parnas72] D.Parnas. *On the Criteria to be Used in Decomposing Systems into Modules.* New-York, communication of ACM, Volume 15, December 1972.
[Pop04] Pop A., Savga I., Assmann U., Fritzen P. *Composition of XML dialects: a ModelicaXML case study.* In Software Composition Workshop Preliminary Proceedings, Barcelona, 2004.
[RECODER] *The RECODER homepage.* http:recoder.sourceforge.net.
[REWERSE] *The REWERSE project homepage*. www.rewerse.net
[Sha96] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice-Hall, 1996.
[Tah97] W.Taha, T. Sheard. *Multi-staged Programming with Explicit Annotation*. New-York, ACM, 1997. Pages 203-217.
[Tar99] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton, Jr. N *Degrees of Separation: Multi-dimensional Separation of Concerns.* In Proceedings of ICSE'99, pages 107-119, Los Angeles CA, USA, 1999.
[Sch98] Scheer, A-W. *ARIS-Business Process Frameworks.* Springer, Berlin, 1998.
[Szy98] Szyperski, Clemens. *Component Software: Beyond Object-Oriented Programming.* ACM Press and Addison-Wesley, New York, N.Y., 1998.
[Wuy01] Wuyts, Roel. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation.* PhD thesis, Vrije Universiteit Brussel, 2001.
[Wuy01+] R. Wuyts and S. Ducasse. *Symbiotic reflection between an object-oriented and a logic programming language.* In Multiparadigm Programming with Object-Oriented Languages, volume 7, pages 81-96. John von Neumann Institute for Computing, 2001.
[Wuy01++] R. Wuyts, S. Ducasse, and G. Arjevalo. *Applying experiences with declarative codifications of software architectures on cod.* In 6th Workshop on Component Oriented Programming (ECOOP'01), 2001.