Beyond Generic Component Parameters

Uwe Aßmann

Research Center for Integrational Software Engineering (RISE),
PELAB, IDA, Linköpings Universitet, Sweden,
`http://www.ida.liu.se/~rise`, `uweas@ida.liu.se`

Abstract. For flexible use in application contexts, software components should be parameterized, but also extended appropriately. Until now, there is no language mechanism to solve both problems *uniformly.* This paper presents a new concept, *component hooks.* Hooks are similar to generic component parameters but go some steps beyond. Firstly, they allow genericity on arbitrary program elements, leading to *generic program elements.* Secondly, they introduce an abstraction layer on generic parameters, allowing for structured generic parameters that bind several program elements together. Thirdly, if they are abstract set or sequence values, they can also be used to *extend* components. Lastly, since they only rely on a meta model they are a language independent concept which can be applied to all languages.
Hooks form a basic parameterization concept for components written in languages with a meta model. For such languages, hooks generalize many well known generic language mechanisms, such as macros, semantic macros, generic type parameters, or nested generics. They also provide a basic concept to realize simple forms of aspect weavers and other advanced software engineering concepts.

1 Introduction

Over time, various generic parameters concepts have appeared in programming languages and component systems. Mainly, they allow for parameterizations of classes, types, or packages with other classes and types. A generic parameter marks one or several program elements in a component which should be replaced consistently by a valid type. Spoken in more abstract terms, the *substitution* or *bind operation* substitutes every reference of a generic parameter type to a reference of a type.

This paper introduces *hooks*, an abstraction concept for generic parameters which generalizes them in several directions.[1] Firstly, hooks provide genericity for arbitrary program elements, not only types (Section 2). Secondly, hooks

[1] Hooks enable us to attach things to other things. In the literature, the metaphor has been used several times to denote parameterizations of components, e.g., of parameterization of classes [Pre95] or extensions of components such as procedure extensions in emacs Lisp [LLStGMG98]. Here, we use the word in a similar way, but relate it to arbitrary meta objects of the component language.

© Springer-Verlag Berlin Heidelberg 2002

may be structured. This allows for structured parameter values that parameterize component parts which are not directly related (Section 3). Thirdly, hooks generalize generic parameters to sets and sequences of program elements. Then, hooks can be *extended* to enrich a component with additional functionality (Section 4). Fourth, many hooks can be regarded as being *implicitly defined* by the programming language (Section 5). This simplifies component extensions. Since the component language can easily be varied, also to XML or binary languages (Section 6), hooks provide a general parameterization and extension mechanism for every language with a meta model.

In essence, the concept of hooks introduces an indirection between the program elements of a component and the actual generic parameter. Hence, hooks introduce a new abstraction level for generic parameters; generic parameters are no longer directly tied to program elements.

2 Generic Program Elements

For this paper, we assume that components are programmed in a strongly typed programming language with a compile-time meta model. Such languages are called *open languages* [CM93] [Aßm98] since they originally are designed for language extension. They support *static meta-programming*, execute the meta programs during compilation, and remove them afterwards. In contrast, modern object oriented languages support a run time meta model (Java, C#) which is not available at compile time (*dynamic meta programming, reflection*).

We start with some basic definitions. Every element of a program corresponds to a language concept. In an open language, the concepts are represented on the meta level, i.e., as types in the language's meta model. Hence, every element of the program is related to a meta object, a type in the language's meta model.[2]

Definition 1 (Principle of Type Safe Substitution). *In a language with a meta model, if a program element should be substituted by another, it can be checked whether the meta model type of the replacing program element is equal to that of the replaced program element.*

We assume that a component is a set of arbitrary program elements.

Definition 2 (Component). *A* component *is a set of program elements.*

This definition of a component is rather general. It covers many cases of static parameterization and static composition of components. Any kind of source code units, such as classes, methods, packages, even aspects may be regarded as components. However, the definition does not cover run time composition.

Definition 3 (Hook). *A* hook *is a set of program elements or positions in a component, being marked-up as generic.*

[2] The literature uses these terms rather loosely; actually it is the language's model and the program's meta model. However, even UML's model is called *UML meta model* although it is a meta model for UML specifications, and a model for UML.

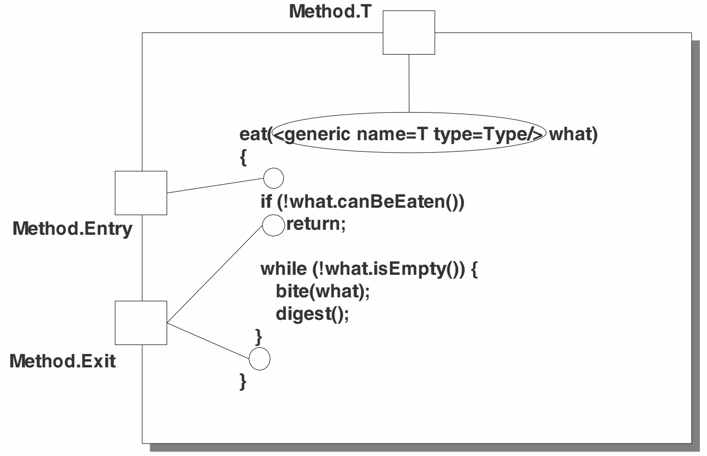
Hooks generalize generic parameters. Their definition will be explained and elaborated on in the paper. As a base language for components, we use Java, although any other language can be used. As a notation for hooks, we use XML markup: Appendix 1 contains an XML schema. Other markup techniques, such as language extensions, can also be employed. In this scenario, hooks are sets of program elements which are marked-up as generic. For instance, a hook that marks up a generic super class looks as follows:

```
class Chicken extends
    <generic name="Super" type="Type"> Animal </generic> { .. }
```

Alternatively, a markup may be *empty* which means that it marks a position in the component. A hook that marks up a super class position looks as follows, using the abbreviation syntax for XML closing tags:

```
class Chicken extends <generic name="Super" type="Type" /> { .. }
```

Figure 1 illustrates that hooks provide an *indirection* mechanism for denoting generic parameters and positions of components. It displays a method component with three hooks: a generic type parameter, a hook for the entry point of the method, and a hook for the exit points of the method. In the case of the generic parameter `T`, the hook `T` marks up a position for a type reference. In the case of the method entry, the hook points to the entry position of the method. This hook is predefined by the programming language and marked up implicitly (Section 5). In the case of the method exit, the hook refers to the two exit positions where control flow returns from the method. Hooks describe generic parts of components in a more abstract way than generic parameters do since they abstract from the concrete representations of the component.



```
eat(<generic name=T type=Type/> what)
{
  if (!what.canBeEaten())
    return;

  while (!what.isEmpty()) {
    bite(what);
    digest();
  }
}
```

Fig. 1. A method component `Method` with three hooks, a generic type parameter, a hook for the entry point of the method, and a hook for the exit points of the method.

We use XML mainly for presentation purposes. Actually, XML markup has a major deficiency since its substitution mechanism is not type-safe, and hence too weak for our purpose. Standard methods for XML transformation, e.g., languages such as XSLT, replace markup with context free pattern matching and disregard the context and *semantic constraints*. Here, we require a type-safe rewriting machinery: a *bind* operation for a generic hook must check whether the value it substitutes for the markup fits into context. Such checks are typically performed by a template expander in a compiler, or by the evaluator of the static meta programs. Here, they are assumed to be a given underlying mechanism.

Example 1. Generic modifiers are generic program elements. For instance, the `synchronized` modifier in Java ensures exclusive execution of a Java method. When they are marked up as being generic several methods can be consistently instantiated for a parallel context:

```
class Chicken extends Animal {
    <generic name="Synch" type="SynchModifier" /> eat() { .. }
    <generic name="Synch" type="SynchModifier" /> drink() { .. }
}
```

Here, `Synch` is the name of the generic modifier hook. During parameterization, its markup can be replaced by the `synchronized` modifier. Suppose, several chickens live in parallel and nurture from a shared food and water resource. Then, this resource should have its access synchronized. In the following example [[ ]] is an operator to read a component from a file, and << >> is an operator to produce program elements from strings:

```
[["Chicken"]].findHook("Synch").bind(<<"synchronized">>);
```

This expression expands the component to:

```
class Chicken extends Animal {
    synchronized eat() { .. }
    synchronized drink() { .. }
}
```

Hence, generic modifiers instantiate several methods consistently to the same synchronization behavior. Also, due to type safety, it can be enforced that generic synchronization modifiers are replaced by synchronization modifiers. In general, generic program elements introduce consistent behavior for different components beyond the purposes which can be modeled by inheritance.

Related Work That Uses Generic Program Elements. To our knowledge, the first language that provided generic parameters was CLU [LAB+79]. Similarly, type parameters were employed in Ada83 [Ada83]. Ada95 generalized the generic type concept to generic packages, packages to which classes can be passed as parameters [Ada95]. This concept allows for easy construction of frameworks since large subsystems can be parameterized by classes. From Ada83, there also leads a trace to C++ templates [Str97]. They are more flexible since parameter values can be concatenated to identifiers. For instance, this allows for the renaming of methods:

```
template class Chicken<class Color> {
   eat<Color>() {...}
}
```

which then is expanded to `eatBrown` if `Color` is bound to `Brown`.

Template Metaprogramming employs C++ templates for more sophisticated purposes, e.g., for static control flow constructs [CE00]. These are control flow constructs which are evaluated at compile time, resembling the `#ifdef` statements of the C preprocessor. However, they are evaluated by the standard template mechanism and provide type-safety. On the other hand, this mechanism represents all generic program elements with generic types, and that might be the wrong way of abstraction.

PARIS provides *program schemes* which can be parameterized by all kinds of program elements [KRT87]. It supports a form of type-safety although it does not yet provide an explicit meta model. Instead, it guesses the type of a generic parameter from its position in the template. Beyond simple substitution, PARIS proposes an parameterization process which is guided by a rule base and produces a software artifact automatically. From the papers, it is unclear how successful this automatic parameterization process has been in practice.

BETA *slots* are generic program elements which may be substituted by *code fragments* [BNS$^+$91] [LKLLMM94]. The substitution is guided by the language's grammar. Only strings that are produced from a certain non-terminal (fragments) may be substituted for the non-terminal. The BETA meta-programming system enforces correct substitution and allows reuse of fragments because they are stored in files. Fragments can even be compiled separately. Hence, BETA slots and fragments are one of the most advanced genericity concepts available.

Semantic Macros extend standard macro processors by letting the macro access results of the semantic analysis of the compiler [Mad89] [KFD99]. Then, macro substitution can be made type safe. Every macro has a result type in terms of the meta model (or, in the types of the abstract syntax tree, if a meta model is not available). Since the context of a macro reference may query the type which is required to be substituted, macro references are rejected if they do not substitute to the right meta model type. Hence, Semantic Macros simplify the use of type-safe substitution and they provide a simple implementation technique for component parameterization. They can even serve as language extensions for hooks, i.e., can embed the hook concept in a component language. However, Semantic Macros are *unstructured* and cannot deal with the structured parameterizations in the next section.

3 Structured Generic Parameters

Since hooks introduce an indirection concept for generic parameters, the parameterizations become more flexible. Furthermore, hooks can be structured to represent *structured generic parameters*. Structured generic parameters provide another degree of freedom for parameterization. Every part of the structured parameter substitutes a different value. Hence, in one go several parameterizations can be performed together, and a component may be parameterized much more flexibly than with unstructured simple generic parameters.

Example 2. As an example, consider a communication between two partners, obeying a communication protocol. Such a protocol must be initialized, usually in a constructor, and finalized, usually in a destructor (we assume for this example that Java contains destructors with the usual syntax of C++ or C#).

We extend the definition of our `Chicken` component (Figure 2). As the XML schema in the appendix indicates, a structured hook can be compared to a record. Every part of a structured hook must name the structured hook (`sname`), must indicate a `feature`, and a `type`. For our component, the following class for the value of the structured hook can be defined:

```
class ProtocolValue { Statement init; Call call; Statement destr; }
```

We can write the following extension program that binds the structured generic parameter with a structured value

```
[["Chicken"]].findHook("SendEggs").bind(new ProtocolValue(
    <<"egg.initialize()">>,<<"human.receive()">>,<<"egg.finalize()">>));
```

To simplify the example, we omit the sub-hooks for the definition of the objects `egg` and `human`. Then the following class results:

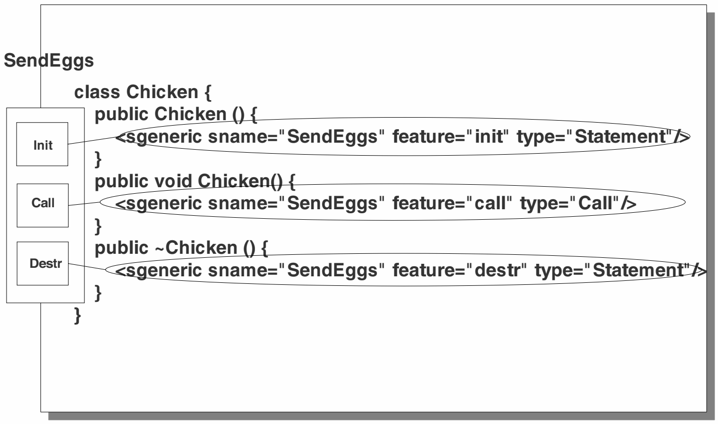

SendEggs
Init
Call
Destr
class Chicken {
public Chicken () {
<sgeneric sname="SendEggs" feature="init" type="Statement"/>
}
public void Chicken() {
<sgeneric sname="SendEggs" feature="call" type="Call"/>
}
public ~Chicken () {
<sgeneric sname="SendEggs" feature="destr" type="Statement"/>
}
}

Fig. 2. A structured hook. It can be replaced in one go with a structured value of parameters. It parameterizes disconnected component parts.

```
class Chicken {
    public Chicken () {
        <sgeneric sname="SendEggs" feature="init" type="Statement">
            egg.initalize();
        </sgeneric>
    }
    public void produce() {
        <sgeneric sname="SendEggs" feature="call" type="Call">
            human.receive(egg);
        </sgeneric>/>
    }     public ˜Chicken () {
        <sgeneric sname="SendEggs" feature="destr" type="Statement">
            egg.finalize();
        </sgeneric>/>
    }
}
```

For the moment, we leave the hook markup in the component, it is to be extended again in Section 4.

Nested hooks provide one advantage over standard parameterization mechanisms such as polymorphic hot spots [Pre95]. The above example can only be expressed by subclassing and polymorphism if all inserted calls go to the same object. If, as in the example, different objects are called, subclassing is not sufficient. However, a structured generic value can provide different values for all of the three sub-hooks.

With structured hooks, several generic parameters of sets of components can be parameterized together in a consistent way. Hence, they lend themselves to generic frameworks. Since we have used a very general component notion these frameworks may be generic over any type of program unit.

Related Work. Program scheme approaches such as PARIS allow to parameterize program parts with several parameters, however do not support structured generic parameters for binding several generic parameters together. One approach with structured generic parameters is GenVoca [BST^{+}94]. GenVoca expresses structuring by nesting and describes nested values with a context free language over the possible values. On the other hand, GenVoca does not allow to markup components, and requires that all subvalues of a structured parameter are substituted to one position in the component. Hence, it does not support parameterization of disconnected program parts.

The only available fully-fledged nested generic parameter mechanism is BETA slots and fragments [LKLLMM94], although it has not been recognized as such. BETA slots (the hooks) are instantiated with BETA fragments (the values), and these may contain slots so that a fragment can nest slots. However, nested fragments must be created as a sequence of parameterizations; a closed form for a parameter value cannot be created. However, this differs only marginally from our approach.

Different program parts can only be parameterized together if the generic parameter *points to* the parts, but is not *identical* to the parts. And structured hooks provide this indirection.

4 Set Hooks for Component Extension

In this section, we consider the case when a hook refers to program elements or positions found in a set or list of equally typed program elements. Many language elements appear in lists or sets, e.g., fields in classes, parameters in parameter lists, or modifiers in modifier sets. With such *set hooks*, it is possible to generalize the notion of a generic parameter to component extensions. If we allow that a hook is bound several times, i.e., if it may be *extended* with additional program element values, a component can be extended step by step by extending of its hooks (Figure 3). We assume a suitable *extend* operation for this purpose which also should be type safe.

Hook extension is useful for many purposes. It goes beyond standard binding of generic parameters because it does not only allow for parameterization, but for extension. Thus it is important for all those situations in component based software engineering when a component based system needs to be extended with new functionality. These situations often occur in software evolution or incremental software processes such as XP [Bec99]. Also, adding a new member to a class is equivalent to the extending the hook of its members. Hence, hook extension can model class extensions in inheritance, as well as merge operations in record and class calculi [Bra92]. However, this only holds for pure extensions without overriding old members. To mimick the full effect of extension and merge oper-

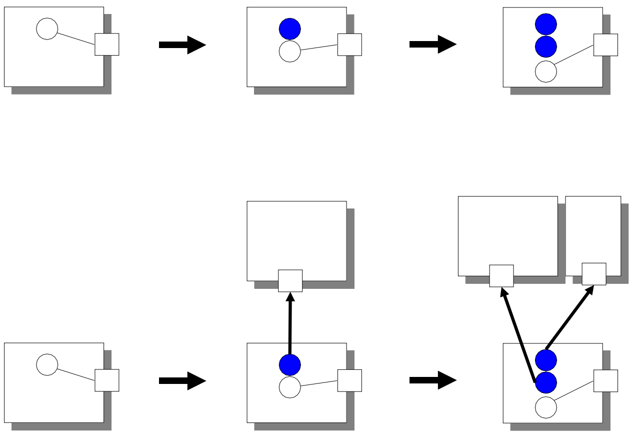


Fig. 3. Extending a hook for component extension. Above: adding more and more program elements to the hook. Down: Adding program elements that relate the component to other components, e.g., communication statements.

ations, the extensions must be checked whether they override existing members.

Example 3. Extensions of hooks can also model simple aspect oriented extensions. In aspect-oriented programming, *aspects extend* a *system core* with additional features [KLM+97]. For instance, an animation aspect can be specified separately and weaved by an extension into the core.

Suppose, we extend the hooks in Example 2 a second time with the following operation

```
[["Chicken"]].findHook("SendEggs").extend( new ProtocolValue(
    <<"initEvents()">>,<<"fireEvent()">>,<<"finalizeEvents()">>));
```

then this example provides a simple form of aspect weaving. The extension operation can be thought of as adding an animation aspect to the core of the component. It introduces new statements which send an event to an animation object, and this additional object animates that the chicken has produced an egg and sent it to the waiting human. This extension can be repeated for other components to be animated, and in this way a useful animation aspect can be added to a system core.

Aspect weavings can be modeled as extensions of hooks if the aspects depend on the core but not vice versa. Spoken in terms of program analysis, there should be forward data dependencies from the core to the aspect, but not vice versa. When extending the core with such forward dependencies, the semantics of the core is not changed, but the aspect receives all necessary data from the core so that it can execute. Clearly, hook extension works for the animation aspect and other ones, for instance debugging or communication aspects. Of course, extension of set hooks has the usual problems of Aspect Oriented Programming, e.g., the aspect interaction problem. Whenever two extensions conflict with each other semantically, the order in which they are applied is important. It is future work to develop precise criteria when these problems occur, how they can be detected, and how they can be remedied.

A generic mechanism that supports aspect orientation must provide nested parameterization. Since an aspect is defined as a concern that *cross-cuts* a core, weaving requires that many parts of the core are extended with different values *consistently.* Until now, no generic mechanism for such extensions was known. Our hope is that aspect weavers can be simplified with extensions of structured hooks, at least for aspects which only depend on the core.

5 Implicit Hooks

So far we only allowed generic parameters which were declared to have a name and a type in the meta model. However, the block structuring rules of a language aid in identifying many *implicit positions* which have standard names. These can be regarded as generic parameters which are *implicitly defined* by the language report. For instance, Figure 1 contains two hooks which have been implicitly defined by the semantics of a method: every method has an entry point, and one or several exit points.

For such *implicit hooks*, we can introduce default names that need not be declared by the component writer, but can be inserted automatically from a component analyzer. Such a tool can mark up implicit hooks without human intervention. Compare Figure 4 with Figure 1. Its markup can be derived automatically if default names for method entry and exit are given.

Example 4. If extensions can address implicit hooks with their default names aspect weavings become simpler. Consider the following extension which extends Example 2 with a debugging aspect. In our simple case, the aspect wraps all methods of the `Chicken` class with additional entry and exit code that prints debugging output:

```
[["Chicken"]].findHook("Chicken.<method>.Entry").extend(
    <<"System.err.println("enterMethod <method>");">>);
```

which results in the following component, still being generic:

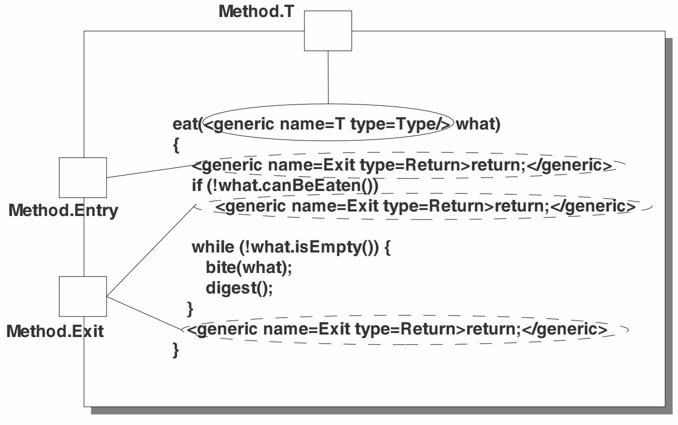
```
class Chicken {
    public Chicken () {
        System.err.println("enterMethod_Chicken()");

        <sgeneric sname="SendEggs" feature="init" type="Statement">
            egg.initalize();
        </sgeneric>
    }

    public void produce() {
        System.err.println("enterMethod_produce()");
        <sgeneric sname="SendEggs" feature="call" type="Call">
            human.receive(egg);
        </sgeneric>/>
    }

    public ~Chicken () {
        System.err.println("enterMethod_~Chicken()");
        <sgeneric sname="SendEggs" feature="destr" type="Statement">
            initalize();
        </sgeneric>/>
    }
}
```

If a component system has a component reader that recognizes and marks up implicit hooks automatically, transformations like the above are possible but component writers need not declare hooks. Since an implicit hook is similar to a join point in Aspect Oriented Programming which is immediately useful for



```
eat(<generic name=T type=Type/> what)
{
  <generic name=Exit type=Return>return;</generic>
  if (!what.canBeEaten())
    <generic name=Exit type=Return>return;</generic>

  while (!what.isEmpty()) {
    bite(what);
    digest();
  }
  <generic name=Exit type=Return>return;</generic>
}
```

Fig. 4. Knowing the programming language semantics and naming conventions, implicit hooks can be derived by a component reader.

aspect weaving. However, it also opens up the component to a larger extent, and sacrifices information hiding. Hence, implicit hooks provide a very weak interface notion, namely the default positions in a component that have been defined by the programming language report. Of course, a component system need not use implicit hooks, or can forbid them for particular components or extensions.

6 Extension to Other Component Languages

Since a markup language treats the underlying language as text the latter can be exchanged. Of course, it can also be exchanged with XML languages. Then, the markup XML schema becomes an XML name space that extends another XML language naturally. What still has to be ensured is type safety, i.e., the transformation facility has to know about the meta model of the XML language and ensure type safe substitution and extension.

Also binary components can be made generic with our approach. A machine language also has a meta model. Of course, this model is semantically not as rich as that of its corresponding source language but lends itself to type-safe substitution, markup, and hook abstraction. All concepts can be transfered: parameterization of all meta objects of the machine language, markup with XML or other mechanisms, grouping of several program parts into structured hooks, set hooks, and finally, implicit hooks. It is obvious that a substitution machinery is required which can handle binary representations.

This insight paves the way for a generic parameterization technology that is independent of the underlying component language. If the substitution machinery and the markup technology is chosen appropriately with the component language, the parameterization machinery does not depend on them and can work for different languages. It may very well be the case that we can build *parameterization frameworks* which work for all programming and specification languages, and which are parameterized with a markup technology, a meta-model, and a type-safe substitution machinery.

7 Implementation

The COMPOST library realizes the component model of this paper for Java. COMPOST consists of two layers. The lower layer is a Java transformation and refactoring engine which can read java components, transform them, and pretty print them again [ALN00]. It also ensures type safe substitution and extension. In the upper layer called *boxology,* components, hooks, and simple bind and extension operations are reified as objects. Components are called *fragment boxes* and provide generalized genericity as outlined in this paper. Since COMPOST is a standard Java library it can be used to write parameterization and extension programs similar to those shown in this paper.

At the moment, COMPOST uses a different markup technology to XML. *Hungarian notation* defines naming schemes for identifiers that convey additional semantics [SH91]. Hungarian notation is also used in other component approaches, e.g., in Java Beans. Using these naming conventions for identifiers, the COMPOST component reader finds declarations of hooks, automatically marks up implicit hooks, and finally checks type-safe substitution with regard to its Java meta model. At the moment, we are extending the concepts to XML as a component language. The goal of this work is to provide a component model for XML documents, and to unify software and document composition in a uniform mechanism (*uniform composition*).

8 Conclusion

This paper has introduced, step by step, several extensions of generic type parameters. Once an indirection concept between the program elements of a component and the generic parameter is introduced (*hooks)*, components can be parameterized more flexibly and also extended (generic program elements, structured generic parameters for grouping of parameter values, extension of implicit hooks). As applications, protocol parameterizations, unforeseen extensions and aspect weavings have been shown. Since the parameterization and extension model is independent of the component language, it shows the way towards a general genericity and extension framework.

References

[Ada83] International Organization for Standardization. *Ada 83 Reference Manual. The Language. The Standard Libraries*, 1983.

[Ada95] International Organization for Standardization. *Ada 95 Reference Manual. The Language. The Standard Libraries*, January 1995. ANSI/ISO/IEC-8652:1995.

[ALN00] Uwe Aßmann, Andreas Ludwig, and Rainer Neumann. COMPOST home page. http://i44w3.info.uni-karlsruhe.de/˜compost, March 2000.

[Aßm98] Uwe Aßmann. Meta-programming Composers In Second-Generation Component Systems. In J. Bishop and N. Horspool, editors, *Systems Implementation 2000 - Working Conference IFIP WG 2.4*, Berlin, February 1998. Chapman and Hall.

[Bec99] Kent Beck. *Extreme Programming Explained: Embracing Change.* Addison-Wesley, 1999.

[BNS$^+$91] Lars Bak, Claus Nörgaad, Elmer Sandvad, Jörgen Linkskov Knudsen, and Ole Lehrmann Madsen. *Software Engineering Environments*, volume 3, chapter "An Overview of the Mjölner BETA System", pages 331–362. Ellis Horwood, 1991.

[Bra92] Gilad Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance.* PhD thesis, University of Utah, 1992.

[BST$^+$94] Don Batory, Vivek Singhal, Jeff Thomas, Sankar Dasari, Bart Geraci, and Marty Sirkin. The GenVoca model of software-system generation. *IEEE Software*, 11(5):89–94, September 1994.

[CE00] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Techniques, and Applications.* Addision-Wesley, 2000.

[CM93] Shigeru Chiba and Takashi Masuda. Designing an Extensible Distributed Language with a Meta-Level Architecture. In O. Nierstrasz, editor, *Proceedings of the ECOOP '93 European Conference on Object-oriented Programming*, LNCS 707, pages 483–502, Kaiserslautern, Germany, July 1993. Springer-Verlag.

[KFD99] Shiram Krishnamurthi, Matthias Felleisen, and Bruce F. Duba. From Macros to Reusable Generative Programming. In U. W. Eisenecker and K. Czarnecki, editors, *Generative Component-based Software Engineering (GCSE)*, number 1799 in Lecture Notes in Computer Science, Erfurt, October 1999.

[KLM$^+$97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopez, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP 97*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, 1997.

[KRT87] S. Katz, C. A. Richter, and K.-S. The. PARIS: A system for reusing partially interpreted schemas. In *Proceedings of the 9th International Conference on Software Engineering*, pages 377–385. IEEE Computer Society Press, 1987.

[LAB$^+$79] B. Liskov, R. R. Atkinson, T. Bloom, E. B. Moss, R. Schaffert, and A. Snyder. CLU reference manual. Technical Report MIT/LCS/TR-225, Massachusetts Institute of Technology, October 1979.

[LKLLMM94] J. Lindskov Knudsen, M Löfgren, O Lehrmann Madsen, and B. Magnusson. *Object-Oriented Environments - The Mjolner Approach.* Prentice Hall, 1994.

[LLStGMG98] Bil Lewis, Dan LaLiberte, Richard Stallman, and the GNU Manual Group. *GNU Emacs Lisp Reference Manual.* GNU Free Software Foundation, for emacs version 20.3, revision 2.5 edition, May 1998.

[Mad89] William Maddox. Semantically-sensitive macroprocessing. Technical Report CSD-89-545, University of California, Berkeley, 1989.

[Pre95] Wolfgang Pree. *Design patterns for object-oriented software development.* Addison-Wesley, New York, ACM press, 1995.

[SH91] C. Simonyi and M. Heller. The Hungarian revolution: A developing standard for naming program variables. *Byte Magazine*, 16(8):131–132, 134–138, August 1991.

[Str97] Bjyrne Stroustrup. *The C++ Programming Language: Third Edition.* Addison-Wesley Publishing Co., Reading, Mass., 1997.

Appendix 1: XML Schema for Hook Markup

```
<schema xmlns="http://www.w3.org/2000/10/XMLSchema"
    elementFormDefault="unqualified" attributeFormDefault="unqualified">
<element name="generic" type="Hook"/>
<element name="sgeneric" type="StructuredHook"/>
<complexType name="Hook">
  <sequence>
    <element name="name" type="string" minOccurs="1" maxOccurs="1"/>
    <element name="type" type="JavaType" minOccurs="1" maxOccurs="1"/>
  </sequence>
</complexType>
<complexType name="StructuredHook">
  <sequence>
    <element name="sname" type="string" minOccurs="1" maxOccurs="1"/>
    <element name="type" type="JavaType" minOccurs="1" maxOccurs="1"/>
    <element name="feature" type="string" minOccurs="1" maxOccurs="1"/>
  </sequence>
</complexType>
<!------ The used fragment of the meta model of Java ---------!>
<complexType name="Type"/>
<complexType name="Statement"/>
<complexType name="Call"/>
<complexType name="SynchModifier"/>
</schema>
```