Meta-programming Composers In Second-Generation Component Systems

Uwe Assmann
Universität Karlsruhe
Institut für Programmstrukturen und Datenorganisation
Postfach 6980 76128 Karlsruhe Germany
assmann@ipd.info.uni-karlsruhe.de

November 18, 1997

Abstract

Future component systems will require that components can be composed flexibly. In contrast to current systems which only support a fixed set of composition mechanisms, the component system should provide a *composition language* in which users can define their own specific *composers*. It is argued for an object-oriented setting that this will be possible by meta-programming the class-graph.

Composers will be based on two important elements. First, they will express coupling by graph-based operators which transform parts of the class-graph (*coupling design patterns*). Second, during these transformations, elementary meta-operators will be used to transform data and code, rearranging slots and methods of parameter-components. Thus during their reuse, components are queried by introspection and transformed by meta-programming.

Composers that use meta-programming generalize connectors in architectural languages. Hence they encapsulate context-dependent aspects of a system, and make components independent of their embedding context. Since meta-programming composers may change behavior of components transparently, meta-programming composers will lead to a nice form of *grey-box reuse*, which supports embedding of components (and classes) into application contexts in a new and flexible way.

Contents

1 Introduction

Since long, software engineers have a beautiful dream: they want to build software from standard parts by composition (LEGO principle). Several authors have claimed that software composition is going to become one of the most fundamental principles for the future software industry, because it supports flexible reuse [NM95b] [Nie95]. However, it is not easy to build software like LEGO: it is not sufficient to reuse components as-is, a software component has to be adapted extensively before it can be embedded into a larger system.

To embed components into a larger system, several aspects are important. First components need to be coupled, i.e. their interfaces (*ports*) need to be linked. Second, during coupling components have to be adapted specifically. Since complex coupling contexts require complex adaptation, often this is a major problem. Third, coupled components need to be encapsulated to the outer world, i.e. new interfaces need to be created. Hence component systems are hierarchical, i.e. composed components can be re-used as components. Last, coupled components need to be configured, i.e. often compositions come in several variants, from which one has to be selected.

Usually components are coupled by hand-programming. To this end commercially available component systems, such as JavaBeans [Jav96] or ActiveX [VN96], offer standardized interfaces of components and coupling interfaces, by which the components can be plugged together. More elaborate component systems (also called *software architecture systems*) such as Darwin [MDK92], UniCon [SDK+95], or ACME [GAO95], offer a limited set of *connectors* by which components can be coupled in an abstract way. Connectors link interfaces (*ports*) of components and arrange for embedding and

control flow among them. The major advantage is that communication- and coupling-oriented aspects of a system are encapsulated into connectors. This enables that components can be programmed independently of their embedding context and improves reuse.

In general, two kinds of connectors can be distinguished. *Primitive connectors* are provided by the programming language or the operating system and comprise mechanisms such as method calls, pipelines, or event signalling. *Composite connectors* should be composed of these and should introduce complex interaction schemes among the components. However, currently composite connectors are only a design concept and cannot be programmed. Shaw admits [SDK+95] that in contrast to primitive connectors

> ... Composite connectors may also appear in these diverse forms, we need (but do not yet have) ways to define them as well.

In this work, it is argued that the ability to program composite connectors will be the key feature of a *second generation component system*. Such a system will provide a general *composition language*, which offers to introspect components, to link ports, to adapt components appropriately, and to hide adaptation to other use-contexts (*transparent coupling*).

Our claim is that the elements of a general composition language can be specified with the help of *static meta-programming*. With meta-programming classes and methods in a class-graph can be introspected, adapted, or allocated.[1] In this paper, meta-programming is used to define *composition operators* over components, so-called *composers*. Composers fall into two main categories: *composite connectors* link components and *encapsulators* encapsulate components to the outer world. Typically composers are *structural meta-operators* which transform the class-graph. During composition, a composer needs to adapt components with *atomic meta-operators* that work on single items of the meta-model.

Meta-operators can be evaluated at system generation time or at run-time. In the first case, the meta-programs are partially evaluated and code is generated. In this work, it is concentrated on this *static meta-programming*, as it is a type-secure programming methodology and leads to faster code. In the second case, the meta-operators modify components dynamically (*dynamic meta-programming*). Although this is more flexible as an unknown number of component-types can be created, it may introduce run-time type errors and the code may run slower. The results of this paper can be applied to dynamic meta-programmed composition as well.

When the atomic meta-operators of the composition language are designed carefully, transparent and consistent integration of components is enabled. A criterion is defined which can be used to check that changes of parameter-components during composition do not disturb the rest of the system. Hence, although code and data of components are modified, old use-contexts never need to be changed explicitly. This is a major step forward towards general software composition, since it leads to *grey-box reuse*, i.e. reuse that extends components *transparent* to old use-contexts.

Since composers may be programmed in variants, applications can be configured by appropriate selection of connectors and encapsulators. When a composer couples transparently, it can be exchanged to its variant without changing the coupled components. Hence meta-programming composers allow to configure a software system orthogonally in two dimensions: both components and composers can be varied independently.

Also component systems and *design patterns* are related. It is shown that a sub-class of design patterns, *coupling design patterns*, can be seen as composers: they are graph transformation operators

[1] Meta-programming always refers to a meta-model. In this paper a class-based object-oriented setting is used and the meta-model forms a type scheme whose instances are the *class-graphs*.

that manipulate the class-graph and modify their parameter-classes. Furthermore, as static meta-programming is used, the design patterns can be expanded automatically to code. Hence this paper gives a systematic method to generate code for a certain class of design patterns.

When composers (connectors and encapsulators) are programmed, architectural styles and environments turn out to be component systems in which the style of defining composers is restricted. Hence our paper supports Nierstrasz in saying [NM95a]:

> Component frameworks essentially define architectural styles in that applications built using the same framework will exhibit similar architectural structure and make use of the same kinds of collaborations between components...
>
> Instead of fixing the architectural styles of application through a particular set of composition mechanisms, component frameworks supported by a composition language will be open ended.

A composition language generalizes current *architectural definition languages (ADL)*. Until now it has not been clear how such a language should look like, and this is what this paper suggests: powerful composers can be programmed to enable grey-box reuse.

The next section presents some examples in which components are coupled by variants of event-based design patterns (section 2). It turns out that these patterns can be seen as composite connectors, which may be specified separately from the components. This rises the need for meta-programming. In section 3 *composers* are defined formally and their tasks in future component-systems are discussed: complex composition (section 3.1), encapsulation, and configuration (section 3.5). It is also shown how composers can be represented by ordinary methods in an object-oriented language that supports meta-programming (section 3.2). In particular, an overview is given on the classes of atomic meta-operators which can be found and define a criterion when destructive changes on parameter-components do not disturb the rest of the system (section 4). Lastly, it is demonstrated that several well-known approaches from the literature are applications of meta-programming composers.

2 Event coupling with meta-programming composers

This section presents several examples how meta-programming composers may look like, in particular composite connectors. The examples are based on a certain class of design patterns, namely those that couple components (*coupling design patterns*) [Tic97] [GHJV94]. These patterns can be meta-programmed, i.e. described with programs on the meta-model.

Event coupling is a flexible method to link components [GHJV94] [SN92]. In an event-based coupling, events are fired by an *event source* component and delivered to a *mediator* context[2] which redistributes them to *event listeners* components. All event source and listener components have to register with the mediator, in order to enable the mediator to distribute events correctly. When new listeners register, or the mediator changes, the behavior of the system changes also. Because parameter-components may not know to whom an event is delivered and from whom an event originates, such a change is transparent to them. This is the reason why event-based coupling is so flexible.

In this paper a component model is used that is similar to the conventional object-oriented model. Whereas usually composition may refer to classes *or* objects, in this work meta-programs are resolved statically so that components are classes. Second, a component is an *architectural item* with different granularity than a class or an object [NSL96]:

[2] Also called *event adaptor*, *event handler*, or *event manager*

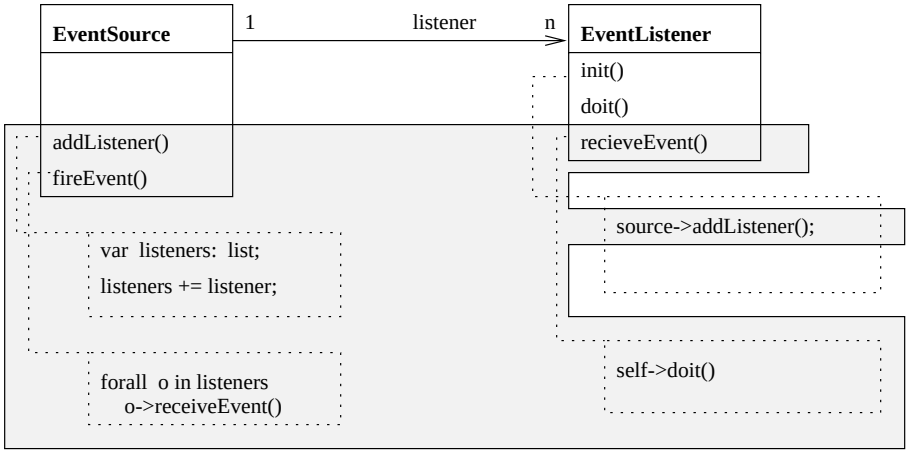

Figure 1: Coupling two components with Observer. Coupling-specific parts are shaded. It is abstracted from the distinction of abstract and concrete classes in [GHJV94].

> Components are designed to be plugged together. Components may be implemented as objects, but they need not necessarily be. The granularity of component is typically coarser than that of objects, but may also be finer. The main difference between the two viewpoints is that the component viewpoint makes system architecture explicit.

Typically, a component consists of a set of classes, and its boundaries vanish in an implementation if it has not been encapsulated explicitly.

2.1 Simple event coupling (Observer design pattern)

The design pattern Observer [GHJV94] provides simple event-based coupling (Figure 1), where the mediator is embedded into event source and the event listener.[3] To this end, the event listener has to provide an interface method `fireEvent` which is called by the event source in case the event occurred. For initialization, the event listener has to register at the event source, calling the `addListener`-method of the event source. Since both need not know statically to whom they will be coupled, the coupling is flexible.

When coupling two arbitrary components with this scheme, both need to be aware of the coupling: The listener has to register himself, and the event source has to maintain a list of listeners to which the events are broadcasted. Thus in Figure 1 context-related parts of the components can be identified that are implemented to enable the coupling. These are depicted in grey shade, whereas context-independent parts are depicted normally.

Suppose a programmer wants to couple components which were not prepared for event-based coupling. Then he has to extend their source code with new parts. Thus in a reuse context, Observer can be programmed only with *white-box reuse*.

JavaBeans event adaptor Components can be coupled more loosely, and Figure 2 shows the design pattern EventAdaptor from JavaBeans 1.0 [Jav96] (also called *mediator* in [SN92]). It is an extension of Observer, and appears in [GHJV94] as the design pattern Adapter in a form that is not

[3] This kind of event coupling was also used in the first version of the Java AWT.

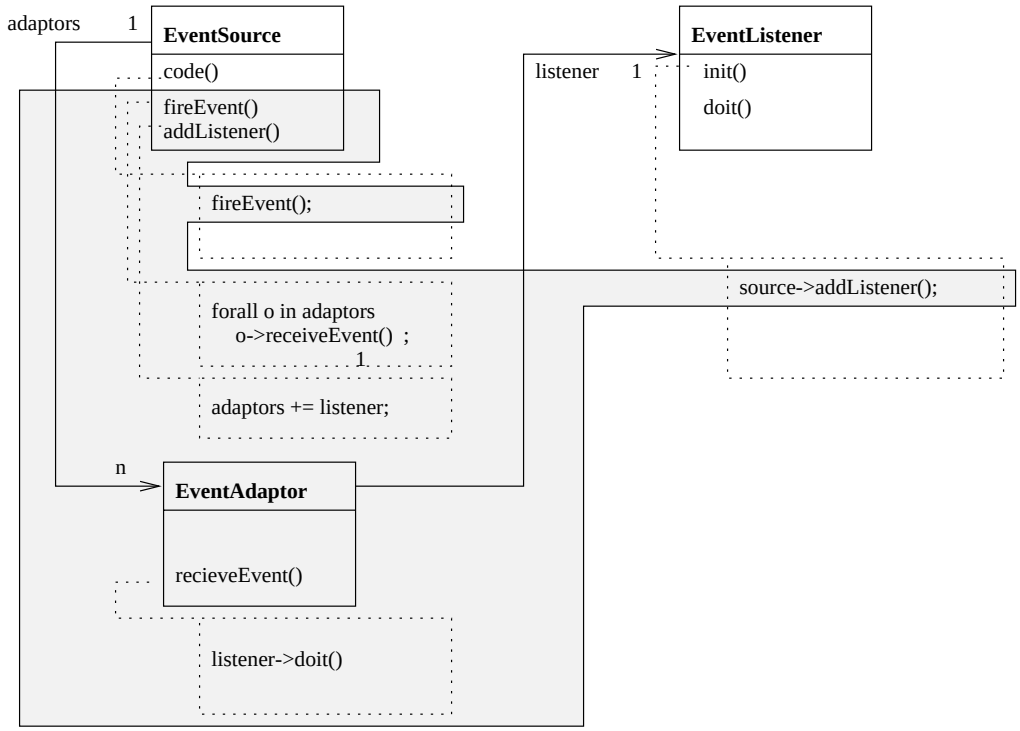

Figure 2: Coupling two components with JavaBeans EventAdaptor design pattern. Coupling-specific parts are shaded. The event adaptor object receives the event and distributes to a listener.

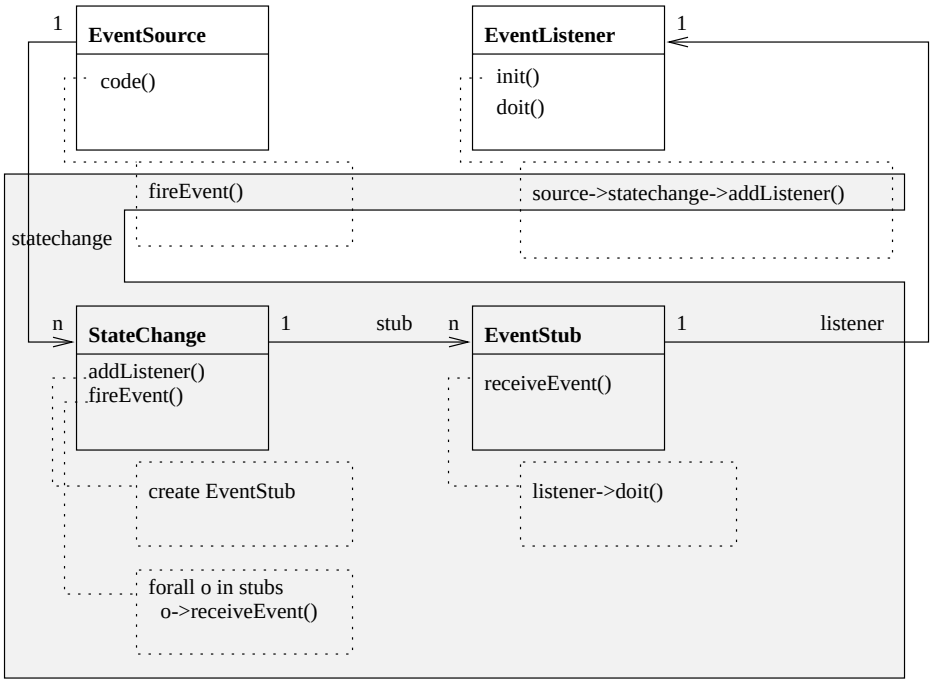

Figure 3: Coupling two components with design pattern EventNotification. Coupling-specific parts are shaded. The listener registration and event signaling is dispersed into *StateChange* and *EventStub* components.

specific to event coupling. In this coupling scheme, the event source maintains a list of listeners to which events are to be distributed. When the source fires an event with method `fireEvent`, an *event adaptor (mediator)* component is called. This adaptor may queue or modify events and has to distributes them to a listener. Hence a listener only has to register for an event at the event source (calling method `addListener`) and to provide an ordinary method that is called when an event occurs (method `doit`). However, since the source needs to do more for the coupling, it will be aware whether it is used in an event context or not. Hence, when components should be reused that were not prepared for event-based coupling, the components have to be extended by event-handling code manually (white-box reuse).

Event notification design pattern Event management can be decoupled from event sources. Figure 3 shows another event coupling design pattern, the EventNotification [Rie96]. Here the management of the listener list is dispersed into a new component, the *StateChange*. At run-time for each event a *StateChange* object is created, and for each listener that registers with an event, the *StateChange* object creates a listener-specific *EventStub* object. These substitute the event-specific listener queue of the event source. When an event occurs, the corresponding *StateChange* object is signalled. As a consequence, all listener-specific *EventStub* objects are called which in turn call the `doit`-routine in the listener.

2.2 From design patterns to meta-programming composers

In the EventNotification, almost all elements of the coupling are separated from the event source and the event listener, except the registration and the event-firing. This introduces a new view on the coupling of source and listener: the context-related parts of the scheme can be regarded as *glue code* which is introduced by a *composer* (in this case a *composite connector*) while connecting the components. In essence, such a composer has to do two things: it has to allocate new components and their interactions (the glue code), and it has to modify the components so that they can be coupled. In this case the composite connector has to create the glue classes *StateChange* and *EventStub* and to mix-in the calls of the methods `addListener()` and `fireEvent()` into the code of the listener and the source. Hence the connector performs *grey-box reuse*: although the user is not forced to edit the component manually (white-box) the component is not reused as-is (black-box), but adapted by mixing-in connector code.

Also Observer and EventAdaptor may be regarded as composers, the only difference is that they couple more tightly and change more details in the parameter-components. The Observer composer does not allocate new components at all, but modifies the code of the components extensively: it has to create the method `addListener()` in the source, the method `receiveEvent()` in the listener, and to add the call to `addListener()` to the method `EventListener.init()`. The EventAdaptor composer allocates a component *EventAdaptor*. While it modifies the event source extensively, it has to mix into the listener only a single call.

Hence an event coupling scheme can be regarded as a meta-programming composition operator, which transforms the class-graph and mixes-in data and code into the parameter-components. Because nothing particular has been required, more design patterns should be realizable by meta-programming composers.

3 Composers: the key to second-generation component systems

This section defines formally how composers and 2nd generation component systems look like. These definitions can be realized directly in a programming language which supports reflection and intercession, and we outline how it would look like in Java.

A second-generation component system has to fulfil the following requirements:

Composition with meta-programming composers Our examples demonstrate that coupling of components should be performed by composers that allocate glue code and modify existing components. This can be done by meta-programming. Any composition action should be *transparent* to the components and should preserve the *consistency* of their code.

Matching and rewriting component connections with structural meta-operators In most cases the interaction of components before the coupling is not as simple as in our examples (in which it consisted only of a simple association). When a composer requires that components are already connected in a certain way (by relations or primitive connectors), these parts have to be *matched* in the class-graph. Thus composers must have a structural matching part whose effect is similar to a left-hand side of a graph rewrite rule. On the other hand, composers have to modify and allocate components, which is similar to the effects of a right-hand side of a graph rewrite rule. Hence composers perform a kind of graph rewriting on the class-graph.

Modifying parameter-components with atomic meta-operators Components need to be adapted by mixing-in code and data. This can be performed by atomic meta-operators that inspect and modify single components.

3.1 Composition with composers

For the definition of composers, a meta-model is assumed which is similar to OMT [RBP+91]. It may be easily extended in case of specific requirements. In this model components are classes. Additional meta-objects are slots, methods, and instructions. Component ports are method calls and method interfaces, and primitive connectors are method calls. Other primitive connectors, such as pipes or events, are assumed to be implemented by method calls. If two components are linked by a method call, they are in use-relation. The other relations between components are inheritance and aggregation. A *class-graph* is an instance of the meta-model:

Definition 1 *A* meta-model $\mathcal{M} = (T, R)$ *is a scheme for a relational graph with a set of labels* $T = \{\text{class}, \text{slot}, \text{method}, \text{instruction}\}$ *and set of relation labels* $R = \{\text{aggregation}, \text{inheritance}, \text{use}\}$.

A class-graph $G = (V_i, E_j)$ *is a relational graph with a family of node sets* $V_i, i \in T$, *and a family of binary relations* $E_j \subset (\mathcal{I} \times \mathcal{I}), \mathcal{I} = \bigcup V_i, j \in R$. $\mathcal{I}$ *is called the set of* meta-objects.

T can be extended for other purposes with arbitrary other meta-objects and relations. Program variables are assumed to be slots of the global context.

The following defines collections of meta-objects:

Definition 2 *A* mo-collection *is a collection*[4] *of meta-objects. A mo-collection of slots is called* data-collection. *A mo-collection of instructions or methods, inclusive the data-collection that is used, is called* code-collection.

A class-graph can be rewritten by a *composer*, an operator that uses graph rewriting and meta-programming.

Definition 3 *An* atomic meta-operator *is an n-ary function* $f : \mathcal{I} \times .. \times \mathcal{I} \to \mathcal{I}$.

A meta-expression E is an expression built from atomic meta-operators. A meta-instruction I is an assignment of a meta-expression to a slot. A meta-program P is a list of meta-instructions.

A composer $C = (L \to R, P)$ *is a complex operator on the meta-model with the following parts:*

- $(L \to R)$ *is a graph-rewrite rule with left-hand side class-graph L and right-hand side class-graph R.*
- *P is a meta-program, performed on the items matched by L.*

An induced subgraph is matched, if together with a set of nodes in the graph *all* incident edges are matched also. Otherwise a *non-induced* subgraph is matched. A composer applies to a class-graph as follows:

Definition 4 *Let G be a class-graph. Then the composition* $G \to_C H$ *with composer C consists of the following steps:*

1. *The user specifies a subgraph* $G' \subset G$ *(the* application point*). G′ is tested, whether L matches it, i.e. a non-induced injective graph homomorphism from L to G′ is found [BFG94].*
2. *The rewriting is performed, i.e. G′ is rewritten to G″ which is injectively homomorph to R.*
3. *The meta-program P is performed on G′.*

[4] Either a set or a list.

The examples from section 2 can be regarded as graph rewrite rules on the class-graph: the left-hand side L consists of the white parts in the figures which are matched in the class-graph. The shaded part, i.e. the allocated glue methods and classes, is introduced by applying the right-hand side R to the matched subgraph. After that the meta-program P modifies the white parts appropriately, i.e. extends the matched meta-objects.

Based on these terms a component systems can be formally defined. A component system is a meta-model with a set of composers:

Definition 5 *A* component system *is a tuple* $\mathcal{CS} = (\mathcal{M}, \mathcal{C})$ *where* $\mathcal{M}$ *is a meta-model and* $\mathcal{C}$ *is a finite set of composers. A* component-based software S *is the result of a sequence* Q *of compositions in the component system, starting from an initial class-graph* Z*:* $Q = Z \rightarrow_{C_1} G_1 \cdots \rightarrow_{C_{n-1}} G_{n-1} \rightarrow_{C_n} S, C_1, \ldots, C_n \in \mathcal{C}$.

Component-based software is the result of a sequence of composer applications. Unfortunately, $\mathcal{CS}$ cannot be an automatic graph rewrite system, since such a system would select arbitrary derivations, and not those the programmer wanted.

3.2 Composers in action

These formal definitions are concrete enough that they translate directly to a textual form in a Java-like style, and in the following the *JavaBeans*-connector from section 2 is demonstrated. All that is needed additionally is a meta-programming interface that provides reflection (querying meta-objects) and intercession (manipulating them). To this end it is assumed that the reflection interface of Java is extended by intercession [KP97].

In such an extended reflection interface, the items of the meta-model, i.e. all meta-objects and -relations, are represented with ordinary classes. The following example uses the meta-object classes `Class`, `Method`, and (implicitly) `Instruction` (Figure 4). We assume some basic reflective methods, such as `findMethod` which finds a method with a name, and `findClass` which finds a class with its name. Also several intercessory methods are required, which form the atomic meta-operators. The operator `new` may also allocate meta-objects, `addMethod` adds a method to a class, `prefix` prefixes the instruction list of a method with some instructions, and `MakeCodeFromText` constructs instruction lists from Java text.

Also the entire component system becomes an ordinary Java class, in which composers are static methods. In a composer $(L \rightarrow R, P)$, the graph rewrite tasks are implemented by matching and manipulating class-graph objects. The left-hand side L is implemented with a list of tests on the parameter-components and their relations. The right-hand side R turns into meta-statements which allocate and link meta-objects. The meta-program P consists of applications of atomic meta-operators and translates directly to a sequence of intercessory method calls.

With composers as methods, users may write programs of composer applications (Figure 5). Suppose three classes `boss`, `assistant`, and `secretary` are given, each of them with a `doit` and `init` method. Before a composer can be applied to a class-graph, a parser has to translate some components from program text to a class-graph. Then the composers (e.g. the EventAdaptor- and EventNotification-connectors) can be applied to the components. Finally, a pretty-printer has to generate Java code which contains the final layout of the classes. Hence complex applications can be plugged together with several calls to composer methods:

Since the composer extends the `doit`- and `init`-methods of the classes appropriately, event communication is introduced automatically by the composer application. Furthermore, since the composer only adds event-firing calls, and these are independent of the old code, the parameter-components are

```
class ComponentSystem {
   public static void ClassGraph parser() {..};
   public static void void prettyPrint(ClassGraph c) {..};
   public static void JavaBeansEventConnector(
       EventSource:Class, EventListener: Class) {
       /* MATCH whether the source and the listener are related by a relation listener */
       if (!member(EventListener,EventSource->listener)) return;
           /* no application possible */

       /* REWRITING: Create meta-objects and meta-object-relations */
       Class EventAdaptor  = new Class("EventAdaptor");
       Method receiveEvent = new Method("receiveEvent",MakeCodeFromText("listener->doit()"));
       Method addListener  = new Method("addListener",MakeCodeFromText(
                                        "for (o = first(EventSource.adaptors);
                                              o != NULL;
                                              o = next(EventSource.adaptors,i))
                                        o.receiveEvent();"));
       addMethod(EventAdaptor,receiveEvent);

       /* MODIFY existing components */
       prefix(findMethod(EventListener,"init"),MakeCodeFromText("source.addListener()"));
       addMethod(EventSource,addListener);
       initialize(EventSource.adaptors);
   }
}
```

Figure 4: A component system as a Java-style class

```
public static void CreateApplication() {
   ComponentSystem cs;
   ClassGraph classgraph = cs.parser();
   Boss boss             = findClass("Boss");
   Assistant assistant   = findClass("Assistant");
   Secretary secretary   = findClass("Secretary");

   /* Compose the classes */
   cs.JavaBeansEventConnector(boss,assitant);
   cs.EventNotificationConnector(boss,secretary);

   cs.prettyPrint(classgraph);
}
```

Figure 5: Composer applications as ordinary method applications

extended *transparently*. This illustrates the power of our approach: components may be programmed independent of their context and, if the added code does not conflict with old code, the components are embedded into the context transparently.

Of course a full-fledged component system would offer a library of composers. Users may use inheritance or even composition to extend them: since composers are components, they can be composed themselves. Composers will be designed along several design dimensions: Which parts of which parameter-components are coupled to others (data-flow dimension)? How complex are links? Which parameter-component executes when (control-flow dimension)? How tight are parameter-components coupled (integration dimension)? Programming composers spans up a large design space of composers, leaving all freedom for users to adapt compositions to their applications.

3.3 Incremental meta-programming

This approach can be extended to a dynamic scenario. If the pretty-printing step is substituted by a code generation step (say to Java bytecode) the generated classes can be loaded dynamically. Additionally, if bytecode can be read and meta-programmed – which is no problem in Java since run-time type information is attached to each class file – the scenario becomes completely dynamic: compositions can be applied during the runtime of a system, the resulting classes are compiled, type-checked, and re-loaded again. Hence *incremental meta-programming* paves the way for incremental dynamic evolution of architectures.

3.4 Aspects of composers

Matching parts of the class-graph and providing glue between components depends on several questions:

1. Which parts of which parameter-components are coupled to others (links)? How complex are links?
2. Which parameter-component is executing when (control flow)?
3. How to couple the parameter-components (degree of integration)?

The answers to these questions span up a design space of composite connectors; in the sequel some of its points are discussed.

Coupling with explicit linking Coupling can introduce linking explicitly, i.e. ports of components are directly or indirectly linked.

with connector-specified control flow Composite connectors can impose control flow ordering between components, e.g. they may start parameter-components in a sequential or data-parallel way, if several work-items can be processed independently. Also composite connectors can set up pipelines of components. In this case primitive connectors should be streams, not only method calls.

without connector-specified control flow If linking of component ports is explicit and control flow of the coupling is not defined by the composite connector, the components call themselves to run. This is the style of the design patterns in [GHJV94] or the architectural specifications in [SDK+95]. In the first case links are represented by method calls; in the latter case by any primitive connector.

Our examples from section 2 are of this kind, as the control flow of the cooperation is determined by events which are signalled by the components. The connector only provides the linking, i.e. which component listens to which events. Also other coupling design patterns, such as Model-View-Controller, or Client-Server, belong to this category.

Coupling without explicit linking Components need not be linked explicitly. Instead components can talk via shared-memory repositories, file systems or databases.

with connector-defined control flow Parameter components need not be linked explicitly, but can be controlled by indirect specification of control-flow, e.g. with rule systems. Then the composite connector creates a rule interpreter which invokes all components according to the rule system.

Examples for this scenario is *make*. Here the components are programs that have to be invoked according to the set of dependencies in a *Makefile*. They communicate via the shared file system. *Odin* is another, more sophisticated example [CO90]. Odin does not maintain explicit dependency files, but calculates the dependency graph from a predefined rule-base and a *control-flow plan*.

without connector-defined control flow Composite connectors may not specify a linking, but turn the parameter-components into active processes, which communicate in a way that is not pre-computable.

Examples for such systems are *coordinating process systems*, invocation by agenda lists (agenda parallelism) [CG89], or *blackboard systems* [GS93]. Because control flow is not predefined, components check themselves when to run.

Distributed vs. tight coupling Also the degree of integration can be determined by the composite connector. If distributed coupling is necessary, connectors may wrap components so that they are dispersed onto different machines. Connectors can generate the necessary marshaling code, proxy components, and service arbiters.

Several languages for distributed systems already provide transparent remote procedure call (RPC) or remote object invocation (ROI) [Ros92] [WJK96]. E.g. CORBA's object request broker seems to be nothing else than a glue arbiter for components.

3.5 Composers encapsulate and configure

Until now only composers have been investigated which couple components. Of course a composer may also abstract its parameter-components into one new component (*encapsulation*). In a component-based software, such hierarchical composition of subsystems is desired, since subsystems hide unnecessary details to the outer world. In our example, this means that after the two event connectors have been applied, an composer should be called that encapsulates the three classes into one component. Such a composer should be formed according to an *encapsulating design pattern*, such as *Facade* [GHJV94]. It would create at least one new component for encapsulation and would link all parameter-components to it.

Configuring a system means to choose one of several variants for some of its parts. When composers can be freely programmed, they can be built in variants, and configuration amounts to selection of composers. Here are some examples:

run-time boundaries Composers may exist in variants that may or may not encapsulate. In the latter case, the component's boundary vanishs in the implementation. Hence exchange of encapsulating composers with their non-encapsulating cousins removes run-time boundaries of components.

control-flow In case that parameter-components can be executed independently, composers may arrange the control-flow of the parameter-components sequentially or parallelly. A sequential composer starts the components in a certain order, a parallel one starts them in parallel and supervises their execution. Hence variant composers lead to scalable software systems: exchange of variants configure a system appropriately for the machine architecture.

Configuration of control-flow is used in the compiler model CoSy for compiler phases [AASv94]: here variants of control-flow-based composers are offered, that run components sequentially or in parallel on a shared memory machine. The code of components need not be changed.

late implementation binding with bridges Late binding of implementations can be expressed by the design pattern of *static bridges* [GHJV94]. Different bridge implementations can be regarded as variant composers which provide a different implementation for a component.

4 Composers adapt components by atomic meta-operators

To adapt parameter-components, a composer has to modify their data slots and method code. This can be performed with several basic meta-programming operators and in this section such a set is discussed. Some operators can be applied both to code and data, as they only require lists or sets of meta-objects.

Atomic meta-operators can be categorized in two classes: *imperative* and *functional* ones. If operators from the former class are applied to components, these are updated *in-place*, i.e. destructively. Of course this may disturb old use-contexts of the component which knew the component in the old shape. Operators of the second class never modify components, but derive new components from old ones. Hence old uses of components are never affected.

Definition 6 *An atomic meta-operator (or a composer) is called* imperative*, if it updates its arguments. Otherwise it is called* functional.

4.1 Harmless and dangerous composition

In program analysis, a *program slice* is a part of a procedure that refers to one variable or one assignment [Wei84] [GL91]. Because a slice encapsulates all statements which are dependent on a variable or a variable definition, the rest of the procedure is independent of it. Two program slices are *orthogonal* to each other, if they are *disjunct*. Then there are no data dependencies between them, e.g. the slices refer to different set of variables and memory locations.

In software composition, code composition is not dangerous, if the composition composes orthogonal slices, i.e. code parts that do not disturb each other.

Definition 7 *Two data-collections are called* orthogonal *to each other, if all names of contained slots are different.*

Two code-collections are called orthogonal *to each other, if they do not have dependencies.*

Theorem 1 *Two code-collections are orthogonal, if they are disjunct program slices. A sufficient condition is that all names of used slots are different and no aliases exist between variables with different names.*

The proof is trivial, and relies on the standard knowledge of program slicing.

Definition 8 *An imperative composition operation on parameter-components is* harmless*, if*

- *it does not delete meta-objects (data or code) of the parameter-component's mo-collections.*
- *all data- and code-collections of parameter-components are orthogonal to mo-collections that are introduced by the composition.*

An imperative composition operator is called harmless*, if it can be used for harmless compositions. Otherwise it is called* dangerous*. An imperative composition operator is called* strongly harmless*, if it can be used only for harmless compositions.*

Harmless compositions retain orthogonal data-collections and orthogonal program slices of parameter-components: newly introduced code does not conflict with old code and does not make old code conflict. Functional composition operators are always strongly harmless.

In a harmless meta-operation, when a code-collection is composed with another code-collection, also data is composed. E.g. when a method is unioned from two instruction collections m_1 and m_2, the object to which the code refers has to get more slots to which the new code may refer. Otherwise m_1 uses slots referenced by m_2 or vice versa, and the program slices are not disjunct anymore.

Theorem 2 *A component c_1 is called* conform *to another component c_2, if c_1 can substitute c_2 in all use-contexts [FZZ95].*

If a component is composed out of parameter-components by a harmless imperative meta-operation, each modified parameter-component is conform to its unmodified version.

Proof: A program slice does not conflict with other code, as it is closed against dependencies [Wei84] [GL91]. Suppose a component c_2 is composed from a parameter-component c_1 which consists of a code-collection m_1. Let U be the set of all use-contexts of c_1. c_1 can only be extended with new code or data.

If c_1 is extended by a code-collection m_2 which is orthogonal to m_1, m_1 will still behave in the same way, since m_1 and m_2 are disjunct program slices. Hence c_2 shows the same behavior as c_1 when it is used in a $u \in U$: of course, both m_1 and m_2 are executed, but since m_2 is orthogonal, c_2 will show the same behavior in the old context as c_1. If c_1 is extended by an orthogonal data-collection m_3, the same holds.

If several parameter-components are composed, m_2 is not allowed to create dependencies between any of them, i.e. all of them must stay orthogonal. This is the case if m_2 is orthogonal to all parameter-components. ■

4.2 Operators that aggregate mo-collections

4.2.1 Aggregation

Aggregation is an atomic meta-operator that unions or concatenates data or code-collections. Aggregation may be commutative. Components can be aggregated directly or indirectly. Direct aggregation of data-collections means that all slots in the aggregated component can be accessed on the same level without indirections. Indirect aggregation corresponds to delegation: the aggregated parameter-components stay separate but are related to the aggregated component. In direct code aggregation

the code is combined directly (by inlining). Indirect code aggregation means that the aggregated component calls the composed code-collections.

A system that can aggregate code directly is the LambdaN-calculus [Dam95]. In particular, constructor methods can be aggregated, i.e. whenever a class is extended, the constructor is extended also.

Wrappers

A special form of aggregation meta-operators are *wrappers*. Prefixing concatenates a mo-collection to the front of another one, while postfixing appends mo-collections. Wrapping combines prefixing and postfixing. These operators can be applied to code and data; in the first case instruction lists are extended, in the second case slot lists are enlarged.

Code wrapping is harmless, if the wrapping mo-collection does not refer to data of the wrapped code-collection, as this would create dependencies. Instead each wrapping mo-collection has to refer to its own data only, which makes it orthogonal to all other code-collections.

Code prefixing operators common to all methods of an object are called *filters* [Ber94] or *layers* [Bos95]. Method wrapping has been used for a long time in Lisp systems for method combination (KEE [KS87], CLOS [DeM93]), because certain algorithms can be composed along the class inheritance dag with before- and after-methods. It can also be used to couple observers to programs, as the wrappers may transfer information to the observer. A specific example is algorithm visualization [Fri97].

Slot access wrappers

A special case of wrapping is wrapping of slot access methods. In concurrent scenarios, where readers and writers compete, access to a slots may be under the control of a synchronization protocol. In particular, the client-server scenario conforms to this scheme. Suppose, several clients compete for a shared server object. Then the access to a server's slot should be controlled by an appropriate synchronization protocol. However, when only one client wants to access the object, no protocol is necessary and access can be granted freely. Typically, only the application context can provide this knowledge. When the protocol methods can be wrapped into slot access methods of the server object, a composer may use its context-knowledge and mix-in the protocol.

Slot access wrapping for synchronization has been used in *generic synchronization policies* [MWBD92] [McH94]. Here *policies* may be defined apart from the class hierarchy, and are mixed-in as appropriate. Another example is the composite connector EventNotification from section 3. It has to modify its parameter-components in order to enable the event coupling. The event source has to be modified at all places which can fire an event. Typically, this happens while setting slot values in internal data. The composite connector can wrap these slot access methods with event-signalling code.

4.3 Other atomic meta-operators

Shadowers for slots

Composite connectors may introduce transactions on components, in the case that several components update data competitively. Sometimes conflicts between concurrent readers and writers can be resolved if data is *shadowed*, i.e. provided in several versions. Then a *shadow operator* should replicate the data under conflict and modify the corresponding code in each competing component such that the private

copy is referenced. Again, shadowing can be introduced by modification of slot access methods: for each component, the method has to refer to the private shadow slot, so that no conflicts occur.

The compiler model CoSy [AASv94] uses a shared repository for communication among components and provides *shadowing* for data structures in the repository. When two competitive components are started, the data under conflict is replicated, and the components work on their private copies. When they finish, a selector function selects one of the shadows. Because the replication is fine-grained (on object slots), shadowing is implemented by modification of slot access methods.

4.3.1 Code aggregation on conjunctive or disjunctive normal forms of first-order formulas

Boolean expressions can be built from first-order predicate expressions in conjunctive or disjunctive normal form. If the formulas are free of side effects, they are orthogonal to each other and can easily be extended by adding new predicate code-collections. In this way class hierarchies can be generated that test preconditions, postconditions and invariants of classes and methods by means of predicate composition [FNZ96].

A special case of this scenario are *decision trees*, to which new branches or leaves can be added orthogonally. *Decision tables* are decision trees, augmented by action code. Because the action code of each branch is only dependent on the predicates of the branch, it is orthogonal to action code of other branches, and decision table branches can be added easily.

4.3.2 Renaming

Complex composition requires an atomic meta-operator that renames meta-objects. Because meta-operators can introspect the name spaces of their parameter-components and mo-collections, a composer can check whether names of meta-objects would conflict during composition. If so, it can rename the conflicting names consistently by several applications of the rename operator.

For instance, renaming plays an important role in the architecture description language Rapide [LKA+95]. In Rapide, methods from different components may be mapped to each other even if they have different names. This is also useful in linking of object files [FPW96].

4.3.3 Overwriting

Another atomic meta-operator which has to be applied carefully is *overwriting of meta-objects*. If applied imperatively, this operation may not be harmless, since updated components can only substitute their original variants if they conform [FZZ95].

4.4 Operators on specifications

This section presents meta-operators that compose code specifications. Either the rules of interpreters are composed, or a generator expands the specification to the actual code of the composed component.

4.4.1 Code aggregation in an interpreted system

In an interpreted system, code is executed by a virtual machine. Interpreters either require that their code is ordered (instruction-sequence-based virtual machines) or unordered (rule-based interpreter).

Naturally code meta-operators can also be applied to the instruction sequence of a virtual machine. Composition of rule-based specifications should be possible e.g. in Prolog, Datalog, term or graph rewrite systems [Ass94], or constraint systems. Additionally, if a rule system consists of rules

with declarative semantics, only infers new knowledge from old knowledge, and does not update old knowledge, its composition operators are strongly harmless. An example is Datalog [CGT89]. Datalog rule systems have unique fixpoints. When new rules are added, the old fixpoint is part of the new fixpoint, i.e. the solution terms of the old fixpoint are a subset of the solution terms of the new fixpoint. Hence composition operators for Datalog rule systems must be strongly harmless.

4.4.2 Attribute grammars

Current attribute grammar tools are modular, i.e. modules of specifications can be set-unioned in order to arrive at the final specification. Union of attribute grammars assembles all attribute equations together which refer to the same non-terminal. Afterwards the attribute grammar evaluator can be used to evaluate the composed attribute equations. Because attribute grammars are declarative, by definition the composed modules are orthogonal, i.e. do not disturb each other. Hence attribute grammar composition operators are strongly harmless.

Attribute grammars come in two classes. A restricted class can be partially evaluated such that the control flow is statically known. This class of attribute grammars is called *ordered* and determining the control flow is called *computation of visit sequences* [Kas80]. The other class of attribute grammars has to be evaluated by an interpreter, but is more general [Far86] [Gro92]. This class can be used in dynamic scenarios, i.e. when an attribute grammar module is fetched from the web and integrated in the existing module set.

5 Some applications of the technology

5.1 Subclassing is a composed atomic meta-operator

Subclassing (inheritance with code reuse) is a functional meta-operator which performs overwriting and aggregation. It is functional, as new classes are derived from old ones; it is overwriting as methods of super-classes can be overwritten; and it is postfixing as the slot list of the super-class is extended by new slots of the inheriting class.

Mixin-based static inheritance [Bra92] explains inheritance as a mixing process between classes, where the mix is performed by several atomic meta-operators (union, overwriting, etc.). *Dynamic mixin-based cloning* carries this over to cloning-based object systems [SCD+]. When inheritance is explained as functional composition over meta-objects, mixin-based inheritance results as a special case.

Interpreting inheritance as composed meta-operator illuminates the semantic differences of inheritance in current object-oriented programming languages. These differences stem from how the inheritance meta-operator is composed from atomic meta-operators. E.g. In Oberon-2 and C++ classes are records, and subclassing appends new record fields to the record field list. Hence subclassing applies a postfixing operator on the slot list. In other languages, such as Eiffel and Sather, slot collections are sets, and subclassing applies set-union as aggregation operation.

Include inheritance in Eiffel or Sather-K [Goo95] allows to reuse classes such that slot lists can be embedded directly. Slots of the included classes are unioned with the current slots and may be renamed in case of conflicts. *Interface inheritance* in Java [GJS96] unions method interfaces, and does not reuse code. This means that method interfaces are distinct meta-objects.

5.2 Transactions

A transaction has an initialization, a main, a rollback and a commit phase. Since components can be composed transparently, a *transaction composer* can allocate the appropriate glue components (transaction supervisor, rollback engine, commit log manager) and insert calls to begin-, rollback- and commit-transaction methods into the components as appropriate. Hence a transaction composer enables components to be used in transactions that were not prepared for this.

5.3 Interface changes (versions)

When a component is extended, its interface may change. However, old use-contexts should see the old interface, i.e. use-contexts should have a specific *view*. Views on interfaces can be implemented as adapters, which encapsulate interfaces that do not fit to all requirements of their uses [GHJV94]. Adaption may rename methods, change format of data, etc.

One example are schema changes in databases: after a change old query code has to be adapted to the new schema. [Sch93] exemplifies this for the database OBST: in this system all object slots are accessed by methods which are generated specifically to the use-context. When a database schema changes, only the generated adapter module has to be re-generated, and the code of the use-context need not be changed.

5.4 Architectural styles are specific composer styles

Architectural styles [GS93] describe component systems that allow only a certain kind of composer:

Repository systems Components are coupled tightly. Synchronization is done by slot access wrapping.

Blackboard systems Composers do not impose control-flow on the components, but they determine themselves when to run, according to the contents of the blackboard.

Pipe-filter systems These systems only allow the unidirectional flow of work packages in pipes. Data in pipes may be copied or in a shared memory repository.

Implicit-invocation systems Only event-based composers are allowed.

Procedure-call systems Only composers are allowed that introduce method call links.

Layered systems A layer is a complex composer that allows tight collaboration within the layer, and introduces interfaces between the layers.

When the description of the architecture is separated from the components and put into composer applications, exchange of composers change the architecture. Also, exchanging one composer style to another changes the architectural style of a system. For instance, it can be easily imagined that a procedure-call system can be turned into an implicit-invocation system: call-composers need to be exchanged to their event-based cousins.

5.5 A hierarchy of composers

When composers are programmed, several categories can be distinguished.

- Black-box reusers. Atomic composers re-use components as-is, i.e. without modifications. This reuse-style is well-known.
- Adaptors. Some composers may not change components, but may adapt them, i.e. by wrapping them with new functionality.

- Transformers. These use static meta-programming to extend components.

Adaptors and transformers appear both in functional and imperative variants, i.e. they may store modifications in new components or directly extend old components. It is claimed that the latter is very important for second generation component systems. Allowing to extend components imperatively and harmlessly will enable *grey-box* reuse, i.e. reuse that introspects and extends components: Adaptors wrap components and transformers extend and modify components in an arbitrary way, provided that harmless meta-operations are performed.

6 Related work

Architectural styles [GS93] describe component systems that allow only a certain kind of composers:

Implicit-invocation systems Only event-based composers are allowed.

Procedure-call systems Only composers are allowed that introduce method call links.

Repository systems Components are coupled tightly. Synchronization is done by wrapping slot access methods with synchronization protocols.

Blackboard systems Composers do not impose control-flow on the components, but they determine themselves when to run, according to the contents of the blackboard.

Pipe-filter systems These systems only allow the unidirectional flow of work packages in pipes. Data in pipes may be copied or in a shared memory repository.

When the description of the architecture is separated from the components and put into composer applications, exchange of composers change the architecture. Also, exchanging one composer style to another changes the architectural style of a system. For instance, it can be easily imagined that a procedure-call system can be turned into an implicit-invocation system: call-composers need to be exchanged to their event-based cousins.

Adaptive programming (ADP) uses graph rewriting on the class-graph and method aggregation. First ADP computes a set of classes to which new methods are added, evaluating a path expression on the class-graph. This corresponds to the evaluation of a Datalog procedure on the class-graph, or an edge-addition rewrite system [Ass94]. In a second step these classes are extended by new methods which are created from a code specification. In essence, ADP is just a form of static meta-programming. It can be regarded as a powerful super-composer, connecting a set of classes with an mixed-in algorithm. Because the set of classes is computed from a Datalog procedure, the ADP-operator can do more than a structural composer, which can match only a fixed number of meta-objects in the left-hand side of its graph rewrite rule.

Aspect oriented programming (AOP) divides programs into *component parts* and *aspects* [Kic96]. Aspects are merged into the components, just as in our approach composers extend components with context-related code. However, AOP relies on a particular aspect language, which describes the coupling, and an aspect weaver, which performs the coupling. Hence for each class of applications new aspect languages and weavers have to be developed. Our approach is more simple, as it only relies on static meta-programming. As in AOP, the composition process can be expanded to code. Our criterion on orthogonal composition allows to exactly determine when a composition is harmless. As this criterion is based on the features of meta-operators and program slices, and not of a special aspect language, it is valid for all application domains and easy to check.

Context relations allow to adapt objects to their context at allocation time [SPL96]. This is similar to the exchange of superclasses at allocation time [Wec97]. However, adapting components and exchanging superclasses is just a special case of meta-programmed composition at allocation time.

Composition-filters [Ber94] [ABV92] and the *layered object model* [Bos95] represent context-related actions of a class by *filters* or *layers* that encapsulate it. Each message that arrives at a class has to cross this set of filters that modify it. However, composing filters (i.e. wrapping code around methods and objects) is a simple atomic meta-operation. Both approaches can model generic synchronization policies, mixin-based inheritance, delegation and context-related adaptation. [Bos97] details this for context-related adaptation. He argues that components need to be adapted flexibly with *superimposition*. This is a modification of the component's interface or implementation by means of a new layer around it. He demonstrates that adaptation by superimposition is more powerful than black-box reuse, and mentions the idea that adaptation can be meta-programming, but does not elaborate on this. However, meta-programming is more powerful than layering: meta-programming can change components deep inside their implementation while layering can only wrap components.

In his thesis [Zim97], Zimmer develops the idea to use design patterns as transformation operators on the class-graph. Zimmer defines a language in which all actions a design pattern involves can be described systematically (pattern matching on the class-graph, transformations of methods, etc.). Although this provided one of the starting points for our work he did not recognize that his language uses static meta-programming.

Code generation from design patterns has been attempted only recently [BFVY96]. Design patterns are described in the form of [GHJV94], with an additional description in a special language COGENT. This macro-based language is expanded by the `perl` interpreter to C++ code. Since the items of COGENT are classes, this approach is static meta-programming, although it has not been described as such. In our work, code generation from design patterns results naturally, since composed classes can be compiled.

7 Conclusion

This work contributes the following results. First, complex composers (connectors and encapsulators) can be developed by (static) meta-programming, using coupling design patterns as transformers on the class-graph. Second, *harmless composition* has been defined which allows to compose components transparently. The meta-programmed compositions generate glue code between components automatically. Meta-programming composers generalize architectural description languages to a general composition language. Hence this work lays the foundation for future component systems, in which context-specific aspects will be encapsulated in complex composers while application-specific aspects will be encapsulated in components. Since both components and composers can be varied orthogonally, reuse will be enhanced enormously.

White-box reuse is too difficult and laborious; black-box reuse is too primitive; *grey-box reuse* is the way to go, and meta-programming composers enable grey-box reuse.

References

[AASv94] Martin Alt, Uwe Aßmann, and Hans Someren van. Cosy compiler phase embedding with the CoSy compiler model. In Peter A. Fritzson, editor, *Compiler Construction, Springer Verlag*, pages 278–293, April 1994.

[ABV92] Mehmet Aksit, Lodewijk Bergmans, and Sinan Vural. An object-oriented language-database integration model: The composition-filters approach. In Ole Lehrmann Madsen, editor, *Proceedings of the 6th European Conference on Object-Oriented Programming*

(ECOOP), volume 615 of *Lecture Notes in Computer Science*, pages 372–395, Berlin, Heidelberg, New York, Tokyo, June 1992. Springer-Verlag.

[Ass94] Uwe Assmann. On Edge Addition Rewrite Systems and Their Relevance to Program Analysis. In Janice Cuny, Hartmut Ehrig, Gregor Engels, and Grzegorz Rozenberg, editors, *5th Int. Workshop on Graph Grammars and Their Application To Computer Science, Williamsburg*, volume 1073 of *Lecture Notes in Computer Science*, pages 321–335, Heidelberg, November 1994. Springer.

[Ber94] Lodewijk M. J. Bergmans. *Composing concurrent objects*. PhD thesis, University of Twente, Enschede, 1994.

[BFG94] Dorothea Blostein, Hoda Fahmy, and Ann Grbavec. Practical Use of Graph Rewriting. Technical Report Queens University, Kingston, Ontario, November 1994.

[BFVY96] F. J. Budinsky, M. A. Finnie, J. M. Vlissides, and P. S. Yu. Automatic code generation from design patterns. *IBM Systems Journal*, 35(2):151–171, 1996.

[Bos95] Jan Bosch. *The layered object model*. PhD thesis, University Twente, 1995.

[Bos97] Jan Bosch. Adapting object-oriented components. In M. Aksit et al., editor, *ECOOP Workshop on Component Systems*, June 1997.

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

[CG89] Nicholas Carriero and David Gelernter. How to write parallel programs: A guide to the perplexed. *ACM Computing Surveys*, 21(3):323–357, September 1989.

[CGT89] S. Ceri, G. Gottlob, and L. Tanca. What You Always Wanted to Know About Datalog (And Never Dared to Ask). *IEEE Transactions on Knowledge And Data Engineering*, 1(1):146–166, March 1989.

[CO90] Geoffrey Clemm and Leon Osterweil. A mechanism for environment integration. *ACM Transactions on Programming Languages and Systems*, 12(1):1–25, January 1990.

[Dam95] Laurent Dami. Functions, records and compatibility in the lambda N calculus. In O. Nierstrasz and D. Tsichritzis, editors, *Object-Oriented Software Composition*, pages 153–174. Prentice Hall, 1995.

[DeM93] Linda G. DeMichiel. An Introduction to CLOS. In Andreas Paepcke, editor, *Object-oriented programming - the CLOS perspective*, pages 3–28. MIT Press, 1993.

[Far86] Rodney Farrow. Automatic Generation of Fixed-Point-Finding Evaluators for Circular, but Well-Defined, Attribute Grammars. *ACM SIGPLAN Notices*, 21(7):85–98, July 1986.

[FNZ96] Arne Frick, R. Neumann, and Wolf Zimmermann. Eine Methode zur Konstruktion robuster Klassenhierarchien. *Softwaretechniktrends*, 16(3):16–23, 1996. Beitrag zur Softwaretechnik '96.

[FPW96] Lisa Spicknall Fruth, James M. Purtilo, and Elizabeth L. White. A Pattern-Based Object-Linking Mechanism for Component-Based Softwae Development Environments. *Journal of Systems Software*, 32:227–235, 1996.

[Fri97] Arne Frick. *Eine Architektur zur Visualisierung von Algorithmen*. PhD thesis, Universität Karlsruhe, 1997. forthcoming.

[FZZ95] Arne K. Frick, Walter Zimmer, and Wolf Zimmermann. On the design of reliable libraries. In R. Ege, M. Singh, and B. Meyer, editors, *TOOLS 17 — Technology of Object-Oriented Programming*, pages 13–23. Prentice Hall, August 1995.

[GAO95] David Garlan, Robert Allen, and John Ockerbloom. Architectural mismatch: why reuse is so hard. *IEEE Software*, 12(6):17–26, November 1995.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, Massachusetts, 1994.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *Java Language Specification*. Addison-Wessley Publishing Company, August 1996.

[GL91] Keith Brian Gallagher and James R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, August 1991.

[Goo95] Gerhard Goos. Sather-K. Report 8/95, Universität Karlsruhe, Fakultät für Informatik, 1995.

[Gro92] Josef Grosch. AG - An Attribute Evaluator Generator. Technical report, Gesellschaft fuer Mathematik und Datenverarbeitung, Forschungstelle Karlsruhe, August 1992. Language Manual.

[GS93] David Garlan and Mary Shaw. *An Introduction to Software Architecture*, volume 1, pages 1–40. World Scientific Publishing Company, 1993.

[Jav96] JavaSoft. JavaBeans^{TM}. http://java.sun.com/beans, December 1996. Version 1.00-A.

[Kas80] U. Kastens. Ordered attributed grammars. *Acta Informatica*, 13:229–256, 1980.

[Kic96] Gregor Kiczales. Aspect-oriented programming. *ACM Computing Surveys*, 28(4), December 1996.

[KP97] Gregor Kiczales and Andreas Paepcke. Open implementations and metaobject protocols. Technical report, Xerox PARC, 1997.

[KS87] R. Kempf and M. Stelzner. Teaching object-oriented programming with the KEE system. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 22, pages 11–25, New York, NY, December 1987. ACM Press.

[LKA+95] David C. Luckham, John J. Kenney, Larry M. Augustin, James Vera, D. Bryan, and Walter Mann. Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, April 1995.

[McH94] Ciaran McHale. *Synchronisation in Concurrent, Object-oriented Languages: Expressive Power, Genericity and Inheritance*. Ph.D. thesis, Department of Computer Science, Trinity College, Dublin, 1994.

[MDK92] Jeff Magee, Naranker Dulay, and Jeffrey Kramer. Structuring parallel and distributed programs. In *Proceedings of the International Workshop on Configurable Distributed Systems*, London, March 1992.

[MWBD92] Ciaran McHale, Bridget Walsh, Seán Baker, and Alexis Donnelly. Scheduling predicates. In O. Nierstrasz M. Tokoro and P. Wegner, editors, *Proceedings of the ECOOP'91 Workshop on Object-Based Concurrent Computing*, LNCS 612, pages 177–193. Springer-Verlag, 1992.

[Nie95] Oscar Nierstrasz. Research topics in software composition. In *Proceedings, Langages et Modèles à Objets*, pages 193–204, Nancy, October 1995.

[NM95a] Oscar Nierstrasz and Theo Dirk Meijler. Requirements for a composition language. In *ECOOP 94 Workshop on Models and Languages for Coordination of Parallelism and Distribution*, volume 924 of *Lecture Notes in Computer Science*, pages 147–161, 1995.

[NM95b] Oscar Nierstrasz and Theo Dirk Meijler. Research directions in software composition. *ACM Computing Surveys*, 27(2):262–264, June 1995.

[NSL96] Oscar Nierstrasz, Jean-Guy Schneider, and Markus Lampe. Formalizing composable software systems – a research agenda. In *1st IFIP Workshop on Formal Methods for Open Object-based Distributed Systems*, 1996. Paris, France. To appear.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modelling and Design*. Prentice Hall, Englewood Cliffs, 1991.

[Rie96] Dirk Riehle. The event notification pattern – integrating implicit invocation with object-orientation. *Theory and Practice of Object Systems*, 2(1):43–52, 1996.

[Ros92] Ward Rosenberry. *Understanding DCE*. O'Reilly & Associates, Inc., 981 Chestnut Street, Newton, MA 02164, USA, October 1992.

[SCD+] P. Steyaert, W. Codenie, T. D'Hondt, K. D. Hondt, C. Lucas, and M. V. Limberghen. Nested mixin-methods in Agora. Number 707, pages 197–219. Springer-Verlag, New York, N.Y. ECOOP '93 - Object-Oriented Programming 7th European Conference, Germany, July 1993. Proceedings.

[Sch93] Bernhard Schiefer. Supporting Integration and Evolution with Object-Oriented Views. FZI-Report 15/93, Forschungszentrum Informatik (FZI), Karlsruhe, Germany, July 1993.

[SDK+95] Mary Shaw, Robert DeLine, D.V. Klein, T.L. Ross, D.M. Young, and G Zelesnik. Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering*, pages 314–335, April 1995.

[SN92] Kevin J. Sullivan and David Notkin. Reconciling environment integration and software evolution. *ACM Transactions of Software Engineering and Methodology*, 1(3):229–269, July 1992.

[SPL96] Linda M. Seiter, Jens Palsberg, and Karl J. Lieberherr. Evolution of Object Behavior using Context Relations. In David Garlan, editor, *4th ACM SIGSOFT Symposium on the foundations of Software Engineering*, pages 46–57, October 1996.

[Tic97] Walter F. Tichy. Classification of Design Patterns. Lecture slides, Universität Karlsruhe, January 1997. http://wwwipd.ira.uka.de/ tichy/entwurfsmuster.html.

[VN96] Steven J. Vaughan-Nichols. ActiveX chases Java. *BYTE Magazine*, 21(6):27–27, June 1996.

[Wec97] Wolfgang Weck. Inheritance Using Contracts and Object Composition. In *WCOP Workshop on component-based systems at ECOOP 97*, June 1997.

[Wei84] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.

[WJK96] Michael Weiss, Andy Jhonson, and Joe Kiniry. *Distributed Computing: Java, CORBA, and DCE*. Open Software Foundation Version 2.1, February 1996.

[Zim97] Walter Zimmer. *Frameworks und Entwurfsmuster*. PhD thesis, Universität Karlsruhe, February 1997.