Towards A Model For Composed Extensible Components

Uwe Assmann

Universität Karlsruhe Institut für Programmstrukturen und Datenorganisation
Kaiserstrasse 12
76128 Karlsruhe, Germany
assmann@ipd.info.uni-karlsruhe.de

Rainer Schmidt

FZI Karlsruhe
Haid- und Neu-Strasse 14
76131 Karlsruhe, Germany
rschmidt@fzi.de

Abstract

Currently software is difficult to extend. When the definition of an item is changed, often all its uses have to be adapted. In this paper we collect several requirements to avoid this effect and sketch a new model for extensible component-based software, *software cocktail mixers*. In essence, the model combines *views* and *operator-based composition*. We argue that uses of components must have an *abstract view* of the component. Subcomponents should only be accessed by name, and the access mapping function should be *compiler-calculated*. To this end software should be composed using *definition composition operators*. However, the definition of the formal model is not yet finished, and we would like to discuss it with others.

Keywords: Software engineering, software extensibility, software maintainance

Workshop Goals: Contacts to other people in component systems and software composition; discussing formal models for software composition

1 Background

One of the authors (Uwe Assmann) has taken part in the development of the architectural environment for compiler construction CoSy [AAvS94]. CoSy provides an architecture model for tightly-coupled repository systems. It is based on a component-connector architecture and allows flexible reuse of compiler components. Active components (*engines*) can be combined with complex connectors (*interaction schemes*) and communicate via a repository of passive data objects. Components can be refined, which enables hierarchical construction of the compiler architecture. Connectors cannot be refined, but there are seven standard interaction types which are used by the system to generate architectural glue code which ties the engines together. This mechanism is very flexible: within a few hours a new compiler can be configured from a ready set of components, which is distinct in optimization behavior, in compilation speed, and memory consumption.

In order to let components work on a shared memory data pool, CoSy provides a powerful *view* concept for engines. A specific interface language, fSDL, is used to describe an engine's view on the intermediate representation of the program. Thus during the development of an engine other engines are invisible. During the composition of the complete compiler all views are merged together to find out at the actual representation of the data and the mapping of the engine views to this representation. From the merged view the fSDL generator generates all modules by which components access the repository. Hence repository objects can be composed and extended very easily: each new view aggregates new items into the data object, while this aggragation is transparent to all other engines. The view merging and mapping process is so powerful that engines in binary form can be reused without recompilation in other compilers, even if they assume a different layout of data objects in the repository.

Our current work is to generalize the composition process in CoSy to general component systems, and we are interested in the formal criteria that enable transparent composition of components, which contain both code and data.

2 Problem: Order hampers extensibility

Current software systems do not allow for easy extension. When definitions of software components (*component-definitions*) are changed, additionally a lot of using parts in the code (*component-uses*) have to be modified. This process is cost-intensive: inconsistencies are introduced during the editing, which can be detected only very late in the change process. Instead we would like to have a software extension method which prevents these hidden and almost untrackable errors, and restricts the extension task only to definitions of components.

One of the main problems that hampers software extensibility is that uses of components have a fixed view how to access subcomponents. Often subcomponents can be accessed by ordered enumeration. Such a total ordering implies that component-uses need to be adapted in source if component-definitions are extended. Second, ordering is necessary because subcomponents are anonymous.

Example 1 Pattern matching has also been applied successfully in functional languages and specification languages of compiler generators. However, when specifications become quite huge (in the order of ten-thousands of lines), experience shows[1] that positional pattern

[1] Our personal experience consists in coding the optimizer generator tool OPTIMIX [Aßm96] with the

matching hampers modular development seriously. When object types in intermediate data structures of compilers are extended by new fields, all pattern matches on the type have to be updated. This is not only annoying, but a severe source of bugs, especially in tools that support wildcard patterns, because with wildcards the number of subpatterns cannot be checked on completeness anymore.

As in CoSy data objects in the repository are composed and aggregated from view specifications, their final layout is only known to the system's compiler and positional pattern matching is not possible. Hence a CoSy tool has to supply pattern match on *named subpatterns*. Thus the specifications can be extended quite easily: pattern matches on objects only need to be changed, when a new subpattern match is necessary. •

3 Position: Compose software with software cocktail mixers

In this position paper we sketch such a generic software extension model. We argue that component-uses must have an *abstract view* of the component. Subcomponents should only be accessed by symbolic names; the actual mapping of the view to the real layout of the component should be left to the compiler. To decouple uses from definitions, software components should be defined by *composition* using *definition composition operators*. Looking at all compositions of component-definitions, the compiler is enabled to calculate the view mapping. When a component is extended, component-uses only have to be recompiled. Since in the model software is constructed by composition and multiple views of components are mixed in order to find out the final shape of the component, we call the model a *software cocktail mixer* and the composition process the mix of a *software cocktail*.

However, the model is current work and not yet stable. We would like to discuss its basics with other researchers in order to find new ideas how to characterize software extensibility and composability.

An *component-definition* is a definition of a software item. It is composed of a name (from domain $Ident$) and a family of subcomponent-sets. D, the *definition-universe*, is the set of all considered component-definitions. Each subcomponent consists of a named attribute value. Attribute domains can consist of code, data, specification items, or names. Components may be defined recursively. We keep several separate subcomponent-sets because these denote different features of the component.

Example 2 Components may correspond to classes, and component-definitions to class definitions. Class attributes and class methods form two distinct subcomponent sets.

```
class person is
   attributes
      name:STR;
      age :STR;
   methods
      reset_name:  STR -> STR;
end;
```

•

Software components are not only defined but also used: A *subcomponent-projection* is a function which projects a component to a subcomponent. An *component-use* is a point in

pattern match tool PUMA

the software system which applies an subcomponent-projection. The set of all considered component-uses is called U, the *use-universe*. More precisely, the subcomponent-projection is a partial function $get : U \times D \times Ident \rightarrow D$ which projects a component-use, an component-definition and a subcomponent name to the subcomponent. Access to subcomponents is only allowed via this projection function.

Because several definitions of a component (with the same name) are allowed, several component-uses may have different *views* of a component. The function *get* realizes the view mapping. It is calculated specific for each component-use by the compiler after looking at all component-definitions and component-uses. The view mapping may be quite simple:

Example 3 Consider as component-definitions class definitions and as subcomponent-sets class attributes. Accessing attributes and methods of classes corresponds to the executing the *get*-function. •

3.1 Definition composition operators

In order to construct more complex components from simpler ones and to extend definitions, our model uses *definition composition operators*.

A *definition composition operator* is an associative operator $\otimes : D \times D \times Ident \rightarrow D$ which combines two component-definitions and produces a new one with a given name. A *view merge operator* is a specific definition composition operator which combines component-definitions with the same name, yielding a new component-definition with the same name.

Composition operators compose component-definitions. They take two components, and add a new one to the definition-universe. Operators merge operand subcomponent-sets to determine the content and layout of the subcomponent-sets of the new component-definition. View merge operators *extend* components. They delete two component-definitions from the definition-universe, and produce a new one with the same name. They are necessary because components may be defined several times, and we have to unite all definitions of a component to a final component layout.

Example 4 A typical example for composition operators is standard inheritance. Classes have two kinds of subcomponents: attributes and methods. Standard inheritance extends a superclass by new attributes and methods, yielding a new subclass. Here we show an example from a fictive object-oriented language where a superclass `person` is extended to a subclass `worker` by adding a new attribute `employer` as well as a new method `get_employer`:

```
class worker extends person is
  attributes
     employer: STR
  methods
     get_employer: -> STR
end;
```

Because the standard inheritance unions attributes and methods of extensions, we can interpret the `extends` keyword as operator with set-union semantics. Then the definition can be re-written in an operator-oriented style:

```
worker = person ∪ workerparts{ attributes{employer:STR},
                                      methods{get_employer:->STR} }
```

where $\cup$ is a structured set-union operator which works in parallel on the subcomponent-sets. •

3.1.1 Composition sequences and view mapping

When component-definitions are combined with composition operators, trees and forests of operator applications result.

Let $O = \{O_1, .., O_n\}$ be a set of composition operators and $D_1 = \{d_1, .., d_n\} \subset D$ a set of definition components. A *composition forest* is a forest of composition operator applications: inner nodes of the trees are composition operators from O, leaves are elements from D_1 and $Ident$. All leaves of the forest from D_1 form the *definition-input* of the composition forest. A *composition sequence* Q_F is a linearization of a composition forest F. Applying a Q_F to D_1 yields a new set of component-definitions D_2, the *definition-output* of Q_F. Thus Q_F is also a function $Q_F : 2^D \to 2^D$. If $Q_F(D_1)$ contains only one component-definition with the same name, it is called *view-resolved.*

In our model compilers have to analyze the specifications to construct a composition forest on the definition universe. In essence this forest is an abstract syntax tree of the source code, containing all relevant component definitions. Since these can be scattered over the entire source code which has to be translated the compiler has to investigate the entire program. Then the definition forest has to be evaluated to find the number and form of defined components (the definition-output). To this end the compiler has to linearize the forest to find a sequential order of all compositions. Of course one linearization is determined by the source code; but there might be other valid ones.

Then the compiler has to analyze the composition forest to determine the component layouts. Because view merge operators remove their operands from the set of considered component-definitions, the output of a composition sequence may contain a different number of component-definitions with the same name. For view-resolved composition sequences, the definition-output is unique for a certain component name.

In a composition forest with view-resolved composition sequence, the compiler can analyze the way from a leave of the forest to its root to calculate an appropriate view mapping for a component-definition. This can be done for all component-uses and visible component-definitions, according to the visibility rules of the language.

Example 5 In our example sequencing means further subtyping from super- to subclasses. If we define

```
driver = worker  ∪  driverparts{ attributes{kind_of_licence: STR} }
```

then the composition forest consists of the tree

$$\cup(\cup(\text{person}, \text{workerparts}, \text{worker}), \text{driverparts}, \text{driver})$$

In case of simple inheritance the inheritance operator on object attributes can be represented by attribute union. To compute the view mapping, the compiler calculates the offsets of the attributes in the object. In case of static binding for methods the value of *get* is the called function address. In case of dynamic dispatch this address is calculated at runtime using the name and type information of parameters. •

3.2 Software cocktail mixers and software cocktails

This leads us to the central definitions in our model of extensible software systems. We regard software extension as a sequence of applications of composition operators in a software composition system.

Definition 1 (Software cocktail mixer) *A* software cocktail mixer *is a system $M = (D, U, O)$ with definition-universe D, use-universe U, and a finite set of definition composition operators O. If all $O_i \in O$ are commutative, M is called* commutative.

A software cocktail mixer provides the framework in which users can combine and extend software by sequences of composition operations. Software is written by composing component-definitions. Software is extended by new applications of view merge operators. Thus the user writes software by mixing composition operators and definitions to a software cocktail:

Definition 2 (Software cocktail) *A* software cocktail *is a system $C = (M = (D, U, O), F, Q_F)$ with definition-universe D, use-universe U, a finite set of definition composition operators O, a composition forest F of operator applications, and a view-resolved composition sequence Q_F. If all $O_i \in O$ are commutative, C is called* commutative.

With software cocktails, because composition forests and linearizations are given, compilers can recalculate view mappings:

Theorem 1 (Recompilation of view mappings) *Let $C = (D, U, O, F, Q_F)$ be a software cocktail. If U, O, or F is extended, all component-uses $u \in U$ need not be edited, only the get-functions needs to be recalculated.*

3.2.1 Commutative composition in software cocktails

If the composition operators are commutative, arbitrary linearizations of composition forests may be chosen for a software cocktail.

Theorem 2 (Uniqueness vs. permutations) *Let $C = (D, U, O, F, Q_F)$ be a commutative software cocktail. Then $Q'_F(D) = Q_F(D)$ for any other linearization Q'_F of F.*

The theorem says that, for software cocktails with *commutative* operators, the application order of the compositions is not important. All linearizations of a composition forest lead to the same final layout of all components. Even more, if subcomponent-sets are unordered, the compiler may choose an arbitrary layout, containing all collected subcomponents.

This influences the calculation of the subcomponent-projection *get*. Because the view mapping is independent of the ordering of composition operations, it can be calculated on any linearization of the composition forest. This is expressed in the following theorem.

Corollary 1 *Let $C = (D, U, O, F, Q_F)$ be a commutative software cocktail. Let $d \in D$ be a component-definition visible at a component-use $u \in U$.*

Then for $f \in Ident$ the value of the subcomponent-projection $get(u, d, f)$ is the same for any other linearization Q'_F of F.

If all component-definitions have the same name, i.e. only views of one component are described, a commutative software cocktail collapses to a finite powerset product lattice on the subcomponent-sets. In such a lattice composition operators are distributive lattice functions, and are combined of basic set functions. The top of the lattice is the view-resolved final layout of the component, the bottom the empty view.

In order to calculate a view mapping a non-commutative software cocktail, the compiler has to take into account the user-specified application order.

Example 6 In case of multiple inheritance the inheritance operator on object attributes can be represented by catenation of attribute lists, together with overwriting of multiply inherited features. As this operation is not commutative, the order in which multiple inheritance is expanded is important for language semantics. •

4 Open questions

- How are conflicts during compositions solved?
- How can operators be classified? Is there a minimal set of operators?
- How can extensibility be better characterized? In particular, we look for the exact conditions under which an extension of a component does not lead to modifications of its use-contexts.
- Is there a formalization of the problem in terms of algebraic specifications (of course operators can be operators of an algebra)?
- How can view mappings be calculated incrementally so that a compiler does not have to investigate the entire program?

5 Related work and conclusion

This paper has presented the sketch of a new model for extensible software, *software cocktail mixers*. If a software description language adheres to its criteria, and a software component is extended, the compiler can adapt use-contexts of this component automatically, and old software need not be modified manually.

Apparently software cocktail mixers are a generalization of *mixin-based inheritance* [Bra92], such that arbitrary operators to compose components are allowed. Additionally they allow to derive important features of the composed software. Also, the model of software cocktail mixers demonstrates in a formal way why object-oriented languages – which access object attributes by access functions – are better extensible than e.g. functional languages which rely on positional pattern matching and argument passing. It is clear that the latter ones will have problems in larger and evolving projects.

Once the model of software cocktail mixers is completed and well-defined, future languages for software systems should be designed so that they fulfil the criteria of the model. Then components (both of specifications and programs) can be composed and extended very flexibly.

References

[AAvS94] M. Alt, U. Aßmann, and H. van Someren. Cosy Compiler Phase Embedding with the CoSy Compiler Model. In P. A. Fritzson, editor, *Compiler Construction (CC)*, volume 786 of *Lecture Notes in Computer Science*, pages 278–293, Heidelberg, April 1994. Springer Verlag.

[Aßm96] Uwe Aßmann. OPTIMIX Language Manual (for OPTIMIX 2.0). Technical Report RT-0195, INRIA Rocquencourt, 1996. Available at http://www-rocq.inria.fr/doc/ or http://i44www.info.uni-karlsruhe.de/~assmann.

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

6 Biography

Dr. **Uwe Assmann** did his PhD thesis *Generation of program optimizations with graph rewriting* at University of Karlsruhe in 1995. As a side action he implemented the optimizer generator OPTIMIX. He has also been involved in the design of the compiler model CoSy which is marketed by ACE b.V. Amsterdam (`info@ace.nl`). Currently he works in architecture and component systems, and is interested in orthogonal composition of software components.

Rainer Schmidt is scientist at the Forschungszentrum Informatik (center for computer science research) at the University of Karlsruhe, Germany. He manages research projects in the area of workflow-management and component-based systems. He received a master in computer science at the University of Erlangen, Germany.