Flexible Type-Safe Linking of Components for Java-like Languages[*]

Davide Ancona, Giovanni Lagorio, and Elena Zucca

DISI, Univ. of Genova, v. Dodecaneso 35, 16146 Genova, Italy
email: {davide,lagorio,zucca}@disi.unige.it

Abstract. We define a framework of components based on Java-like languages, where components are binary mixin modules. Basic components can be obtained from a collection of classes by compiling such classes in isolation; for allowing that, requirements in the form of type constraints are associated with each class. Requirements are specified by the user who, however, is assisted by the compiler which can generate missing constraints essential to guarantee type safety.

Basic components can be composed together by using a set of expressive typed operators; thanks to soundness results, such a composition is always type safe.

The framework is designed as a separate layer which can be instantiated on top of any Java-like language; a prototype implementation is available for a small Java subset.

Besides safety, the approach achieves great flexibility in reusing components for two reasons: (1) type constraints generated for a single component exactly capture *all possible* contexts where it can be safely used; (2) composition of components is not limited to conventional linking, but is achieved by means of a set of powerful operators typical of mixin modules.

1   Introduction

It has been argued that the notion of software component is so general that cannot be defined in a precise and comprehensive way [12]. For instance, [20] provides three different definitions, that adopt different levels of abstraction. However, most researchers would agree that the following features are essential prerequisites for component technology: *modularity*, *type safety*, and *independence* from a particular programming language.

Modules and components share several common characteristics. The important software engineering principle of maximizing cohesion and minimizing dependencies of code applies as well to modules and to components. Furthermore, both modules and components are meant as units of composition which can be developed independently.

Type safety is an important property which guarantees a correct integration between components; separate development of components requires explicit interfaces not only for the provided services, but also for the requirements which ensure safe assembly of components. In order to maximize reuse, required interfaces should capture as many as possible contexts where a component can be safely used.

While modules are often tied to a specific programming language, components are usually meant as binary units, and therefore should not depend on a particular language; of course, basic components still need to be constructed by using some language. For instance, .NET assemblies do not strongly rely on any particular language, but can be created, for instance, from both C# and Haskell code. However, assembling components is a process which should involve only binary units and, therefore, is expected to be language independent. The benefits of this independence are a better integration and interoperability of components, especially when the binary form is some kind of intermediate language.

Among the several varieties of modules which can be found in programming languages or have been proposed in literature, *mixin modules* are one of the closest approximations of the notion of software component.

Module systems based on the notion of mixin module offer a framework largely independent from the core language with well-established and clean foundations [7, 6, 14]. Differently to parametric modules, like, for instance, ML functors, which offer only one composition operator roughly corresponding to function application, mixin modules are equipped with a richer set of operators that support *mutual recursion* across module boundaries and declaration of *virtual* entities which can be redefined via an *overriding* operator. For this reason, mixin modules seem a good starting point for defining a language independent framework for flexible composition and reuse of components in a type safe way. The main difference between a mixin module and a component is that the former is modeled as a collection of classes in source form, while the latter is modeled as a collection of classes in binary form. Of course, in practice there are other differences which we deliberately do not model in this paper: for instance, in general a component is a collection of more heterogeneous entities including not only code, but also resources like, for instance, multimedia data.[1]

Nowadays component technology is mainly based on mainstream object-oriented languages; nevertheless, object-oriented languages alone fail to provide important features for developing and assembling components. Compositional compilation is not supported by mainstream object-oriented languages, even though this property is important for allowing separate development of components: users should be able to obtain a basic component from a collection of classes by simply compiling such classes in total isolation. Furthermore, linking is the only available mechanism for manipulating and assembling binary components.

In this paper, we investigate how to build a framework for component-oriented programming based on Java-like languages. The framework is meant as a logi-

---

[1] We refer to [20], Section 4.1.4, for more details.

cally separate layer constructed on top of the Java-like language used for creating basic components.

In the framework, components are modeled as mixin modules in binary form, by following and further developing the approach presented in [5]. Furthermore, separate development of components is possible by adopting the type technology we have developed for Java-like languages in a previous work [2]. Thanks to this technology it is possible to specify the minimal requirements needed by a component for being safely used by a set of *polymorphic type constraints*. Compilation in total isolation of classes into components is supported by the notion of *polymorphic bytecode*, a bytecode annotated with type variables which can be instantiated according to the context where a component is deployed.

The framework allows separate compilation of classes into basic components starting from the declarations of such classes in a Java-like language and from the specification of the requirements needed by the classes. Then, components in polymorphic bytecode can be assembled together in a type safe way by means of five composition operators: *bind*, *merge*, *renaming*, *unbind*, and *restrict*.

Other interesting features of the framework are the following:

- Since specifying the requirements needed by a class can be a tedious activity, the framework assists the programmer by generating those constraints which have not been explicitly specified by the user, but are nevertheless necessary for guaranteeing a type safe composition. The interface obtained in this hybrid way is then permanently associated with the polymorphic bytecode of the class in the components.
- Classes in a component are all implicitly considered *virtual*, that is, their definition can be later replaced when composing the component with others.
- In addition to composition operators typical of mixin modules [7, 6], the framework provides two novel operators[2] *bind* and *unbind*, designed for better supporting unanticipated software evolution.

The paper is organized as follows. Section 2 is a gentle introduction to the framework; some examples are used for explaining its main features and its ability to support software reuse and unanticipated software evolution. In Section 3 we formally define syntax and reduction semantics of the framework, by listing the ingredients the underlying Java-like language should provide. Section 4 is devoted to the implementation of the framework: a prototype is available[3] for testing all the examples shown in Section 2. Finally, Section 5 outlines related work, summarizes paper contribution and draws directions for future developments.

A preliminary presentation of the ideas developed in this paper can be found in [3]. The full formal definition of the framework can be found in [4], notably including the definition of the type system modeling compilation of component expressions into binary components and soundness results. Moreover, [4] provides the formal description of an instantiation of the framework on top of Featherweight Java [15], and more examples.

---

[2] Which, however, can be encoded in lower-level operators of module calculi such as CMS [6].

[3] `http://www.disi.unige.it/person/LagorioG/SmartJavaComp/`

2 A Gentle Introduction to Components

This section is a brief introduction to our component-oriented system: its main features are presented through some simple, but still meaningful, examples showing its expressive power. A more involved example showing how to deal with the classical *expression problem* (or *extensibility problem*) [21] can be found in [4]. Even though our operators handle components in binary form (more precisely, in polymorphic bytecode), in the examples we write components in source format for readability. In particular, we choose Java as source language, but all code could be easily rewritten in, say, C#.

2.1 Basic Components

Let us start our introduction with an example[4] of declaration of basic component:

```
component LinkedList = {
  deferred class N;
  class List {
    requires { N(N); }
    N first;
    void addFirst() {
      first=new N(first);
    }
  }
  class Node {
    requires { & N; }
    N next;
    Node(N n) { next=n; }
    N getNext() { return next; }
  }
}
```

A basic component is a collection of declarations of classes which are either *deferred*, that is, whose definition has to be imported later, like N, or *defined* inside the component, like `List` and `Node`. Class definitions are those in the Java-like language under consideration, enriched by a `requires` part which specifies *type constraints* on deferred classes, which of course also depend on the language. In the example, constraint `N(N)` means that class N is required to have a constructor applicable to an argument of type N, whereas constraint `&N` means that class N must exist. Other forms of constraints are subtyping constraints and constraints requiring a class to have a field of a certain type or a method applicable to certain argument types; moreover, constraints are *polymorphic* in the sense that types can be type variables, as will be illustrated below.

As it will be shown, deferred classes can be bound to a definition by means of the *bind* and *merge* operators. Within this example, the intuition is that N *could be*

---

[4] For simplicity, we will avoid access modifiers.

`Node`; indeed, if we replaced all occurrences of `N` with `Node`, then we would obtain the classic example of single-linked lists with a header node. However, having used a deferred class instead of the already defined class `Node` allows us to bind `N` to something more specific than `Node` later, for instance a class `DoubleNode` (which, presumably, extends `Node`).

This particular use of a deferred class allows one to simulate the idea of type *mytype* [10], or `ThisClass` of LOOJ [9], where inside a class, say `Node`, we can use *mytype* instead of `Node` with the effect that in any subclass of `Node`, say `DoubleNode`, this type will be interpreted by `DoubleNode` rather than `Node`.

However, our approach allows a step further: `N` can be bound to *any class* that satisfies the type constraints declared in class `List` and `Node`. For instance, class `Node` simply requires an existing declaration for `N`, since `N` is used in `Node` only as a type, while the correctness of `List` relies on a stricter constraint[5] asking `N` to provide a constructor which takes an argument of type `N` (hence, with a single parameter whose type is a supertype of `N`).

Note that constraints are declared at the level of each class definition, rather than at the level of the component declaration. As we will see, this is due to the fact that classes declared in components are all *virtual:* for instance, a new component could be derived from `LinkedList` by overriding the declaration of `Node`. In this case, the constraints associated to `Node`, and only those, are analogously replaced.

Component `LinkedList` supports an important feature for promoting component-oriented programming: each class is explicitly equipped not only with the interface of the provided services, (what is usually, and improperly, called the provided interface), but also with the interface of the required features. (what is usually, and improperly, called the required interface). Indeed, provided and required interfaces for classes `List` and `Node` can be easily extracted from their code:

```
class List {
  requires { N(N); }
  provides { N first;
             void addFirst();
  }
}
class Node {
  requires { & N; }
  provides { N next;
             Node(N n) ;
             N getNext() ;
  }
}
```

Providing the required interface should allow compilation of a component in *total isolation* (no other sources or binary files are needed) and composition with other components (already in binary form) in a *type safe* manner. To this end,

---

[5] Indeed, the constraint `N(N)` implies `& N`.

the required interface should specify, on the one hand, all the requirements on deferred classes which are needed to compile the component; on the other hand, it should not specify requirements which are not strictly necessary, in order to allow safe composition with as many other components as possible. For Java-like languages, this can be achieved by using the approach we propose based on type constraints, whereas cannot be achieved by using other forms of required interfaces. For instance, compilation in isolation of the component above cannot be achieved by using the approach based on only subtyping constraints adopted for Java generics [8]; there is no way to guarantee that class N has a constructor which is type compatible with the call in method `addFirst` by simply requiring class N to extend some already defined class or interface.

Conversely, an approach where the required interface has to specify for each deferred class its expected signature (that is, constructor, field and method signatures), as done, e.g., in our previous work [5], is too restrictive, since it rejects components which do not match this type but can still be linked in a safe way with the given component. We will illustrate better this point in the following when introducing the merge operator.

Since specifying required interfaces by listing all the needed type constraints may be a tedious and error prone activity, the specification of required interfaces is assisted by the compiler: the most general constraints which are required by a component, but are not explicitly specified by the programmer, are automatically generated and added to the required interface. In this way the compiled code will contain the complete required interface, including both the user constraints and the missing ones inferred by the compiler. Of course, the user can always specify constraints which are not strictly necessary to guarantee type safety, but that are needed for contractual reasons.

For instance, in class `List` the user could specify the requirement `N <= Node` which requires N to be a subclass of `Node`, even though this condition is not necessary for the type safety of the code of the class. However, the required interface generated with the code will contain both the user-defined constraint `N <= Node` and the inferred constraint `N(N)`.

As shown in the following, the generated required interface will be used together with the provided interface, to check type safety of component composition.

2.2 Open and Closed Components

A component with deferred classes, as `LinkedList`, is called *open*; analogously, a component with no deferred classes is called *closed*. Classes declared inside an open component, as `List` and `Node`, cannot be accessed through qualified names (see 2.5).

```
LinkedList.List l=new LinkedList.List(); // type error
```

6

The qualified name `LinkedList.List` is used for denoting class `List` at component `LinkedList`.[6] An unqualified class name is called a *simple class name*. A *soft link* to a class is any of its unqualified occurrences except those which introduce the declaration of either the class itself, or any of its constructors. Analogously, qualified occurrences are called *hard links* (see more in Section 2.5).

There are two different composition operators for deriving closed components from open ones: *bind* and *merge*.

Bind A closed component can be obtained by binding the deferred classes of some open component to definitions in the same component. For instance, a new component `ClosedLinkedList` could be obtained from `LinkedList` by binding `N` to `Node`, since class `Node` satisfies all required constraints on `N`:

```
component ClosedLinkedList=bind(LinkedList,N->Node);
```

The component we obtain in this way is equivalent to (that obtained compiling) the following, where we have copied the definition of `LinkedList` and replaced each occurrence of `N` by `Node`.

```
component ClosedLinkedList = {
  class List {
    requires { Node(Node); }
    Node first;
    void addFirst() {
      first=new Node(first);
    }
  }
  class Node {
    requires { & Node; }
    Node next;
    Node(Node n) { next=n; }
    Node getNext() { return next; }
  }
}
```

Now classes `List` and `Node` can be used:

```
ClosedLinkedList.List l=new ClosedLinkedList.List();
```

When closing a component, all type constraints in the class types must be verified, otherwise a type error is issued.

---

[6] For simplicity, we use here the dot notation for qualified class names since it is likely the most natural choice for Java programmers. However, while this poses no ambiguity problems for the simple Java subset we have implemented so far, this would be the case in an extension to full Java.

For instance, the expression `bind(LinkedList,{N->List})` is not type correct, since `List` does not satisfy the constraint `List(List)`.

Note that the constraints in `ClosedLinkedList` cannot be removed by the compiler even though they are clearly satisfied. Indeed, a closed component is not permanently "sealed", but can be reopened using operators *restrict* and *unbind*, which will be discussed in Section 2.4.

Merge Assume we want to extend the code in `LinkedList` in order to support doubly linked lists. This extension can be isolated in a separate component:

```
component Double = {
  deferred class N, List, Node;
  class DoubleList extends List {
    requires { N(N,N);
               'a List.first;
               N<='a;
               'a N.next;
               'a 'a.prev; }
    N last;
    void addLast() {
      N n = new N(last, null);
      if (first==null) first = n;
      if (last!=null) last.next = n;
      last = n;
    }
    void addFirst() {
      N n=new N(null, first);
      if (first!=null)
        first.prev = n;
      first = n;
      if (last==null) last=n;
    }
  }
  class DoubleNode extends Node {
    requires { Node(N); }
    N prev;
    DoubleNode(N n) { super(n); }
    DoubleNode(N p,N n) {
      super(n); prev=p; }
    N getPrev() { return prev; }
  }
}
```

Before explaining how the merge operator behaves, let us focus on the user requirements in `DoubleList`: the type variable `'a` is used for expressing the

general[7] requirement that class `List` must provide the field `first` with a type `'a` such that `'a` is a supertype of `N` (`N<='a`), and provides a field `prev` having the same type `'a` (`'a 'a.prev`). Note that, as anticipated above, we could not achieve the same effect by using a required interface which specifies for each deferred class its expected signature. Indeed, in this case we should have fixed for instance the type of field `f` in `List`, e.g., requiring this type to be `N`, whereas in fact any supertype of `N` would work as well.

A new component `DoubleLinkedList` can be defined by merging `LinkedList` with `Double`:

```
component DoubleLinkedList=merge(LinkedList,Double);
```

In `DoubleLinkedList` the two deferred classes `List` and `Node` of component `Double` are bound to the corresponding classes declared in `LinkedList`, whereas class `N` remains deferred (indeed binding of deferred classes is by name matching). Note that, while it is possible to merge components with deferred classes having the same name, name conflicts for defined classes are not allowed.

Finally, it is possible to bind `N` to `DoubleNode` in `DoubleLinkedList`:

```
component ClosedDoubleLinkedList = bind(DoubleLinkedList,N->DoubleNode);
```

2.3   Renaming Facilities

Since binding of deferred classes is by name matching, a renaming operator might be useful in some circumstances.

For instance, if in `Double` the two deferred classes `List` and `Node` were named `L` and `Nd`, respectively, then a renaming would be necessary before merging `LinkedList` with `Double`.

```
component DoubleLinkedList =
                merge(LinkedList,rename(Double,{L->List,Nd->Node}));
```

The rename operator allows renaming of a single class name at time, therefore the expression `rename(Double,{L->List,Nd->Node})` is just a convenient shortcut for the more verbose one:

```
rename(rename(Double,L->List),Nd->Node)
```

Renaming of more classes is accomplished sequentially from left to right. Both deferred and defined classes can be renamed. Since the operator allows only bijective renamings, the newly introduced name must be unused in order to avoid conflicts.

---

[7] For sake of simplicity we have omitted to specify the most general requirements as they would be inferred by the compiler.

2.4 Unbind and Restrict

Let us consider again component `ClosedLinkedList` as defined in Section 2.2. As already noted, the constraints on class `Node` cannot be removed by the compiler without compromising type safety. This is due to the fact that it is possible to derive an open component from a closed one by making some class deferred. This can be accomplished by using either the *unbind* or the *restrict* operator.

Unbind The unbind operator can be considered the inverse of bind; for instance, as `ClosedLinkedList` could be derived from `LinkedList` with the bind operator, the opposite could be obtained by deriving `LinkedList` from `ClosedLinkedList` with the unbind operator.

```
component LinkedList=unbind(ClosedLinkedList,Node->N)
```

The class to be unbound (`Node` in the example) must be defined in the component while the new name (`N` in the example) must be unused. The effect consists in adding the deferred class `N` and replacing all soft links to `Node` with `N`.
This example shows also that in general requirements cannot be safely removed by the compiler; indeed, requirements on `Node` specified in `ClosedLinkedList` cannot be simplified, since after applying the unbind operator, soft links to the defined class `Node` could be redirected to some deferred class (`N` in the example). The unbind operator offers an effective way to deal with unanticipated code modification due to poor component design; although unanticipated code modification should be better addressed when designing and developing components, unbind gives a chance to recover from this problem when components are assembled and are not available in source form.

Restrict The restrict operator provides another mean for opening closed components. It is mainly used jointly with the merge operator to override class definitions. For instance, a new component could be obtained from `ClosedLinkedList` by overriding the definition of `Node` with that contained in component `AnotherNode`:

```
component AnotherNode = {
  class Node {
    Node next;
    int elem;
    Node(Node n) { next=n; }
    Node(Node n,int e) { next=n; elem=e; }
    Node getNext() { return next; }
    int getElem() { return elem; }
  }
}
component ClosedIntLinkedList =
            merge(AnotherNode,restrict(ClosedLinkedList, Node));
```

First, the restrict operator makes class `Node` in `ClosedLinkedList` deferred by removing its definition. Then the new definition of `Node` in `AnotherNode` is added by the merge operator.

Note the difference between the unbind and the restrict operator: for class `C` defined in component `Comp`, `unbind(Comp, C->U)` does not remove the definition of `C`, but redirects soft links to `C` to an unused class `U`; `restrict(Comp,C)`, instead, makes class `C` deferred by removing its definition, but does not redirect soft links to `C`. Hence `rename(restrict(Comp,C),C->U)` is still different from `unbind(Comp, C->U)` since in the latter the definition of `C` is kept.

As for renaming, convenient shortcuts are provided for unbinding and restricting multiple classes.

2.5 Qualified Class Names

As already explained, references to classes defined in other components are allowed by using qualified class names:

```
component AnotherList = {
  class List {
    requires { AComponent.Node(AComponent.Node); }
    AComponent.Node first;
    void addFirst() { first=new AComponent.Node(first); }
  }
}
```

Component `AnotherList` directly depends on component `AComponent` which is expected to define a class `Node` satisfying the constraint specified in class `List`. While soft links can always be redirected by the composition operators, hard links cannot and establish direct dependencies between components. However, these dependencies are always made explicit by the required interface. The same consideration applies to hard links to classes declared in the same component.

```
component YetAnotherList = {
  class List {
    requires { YetAnotherList.Node(YetAnotherList.Node); }
    YetAnotherList.Node first;
    void addFirst(){first=new YetAnotherList.Node(first);}
  }
  class Node{
    requires { & YetAnotherList.Node; }
    YetAnotherList.Node next;
    Node(YetAnotherList.Node n){next=n;}
    YetAnotherList.Node getNext(){return next;}
  }
}
```

In component `YetAnotherList` all hard links to `Node` are permanently bound to the definition of `Node` in the same component and can no longer be unbound.

11

While it is not possible to transform a hard link into a soft link, the opposite can be achieved via the bind operator. For instance, `YetAnotherList` could be equivalently obtained from `ClosedLinkedList`:

```
component YetAnotherList =
                bind(ClosedLinkedList,Node->YetAnotherList.Node);
```

3   A Framework of Components

The full formal definition of the framework of components we have introduced through examples in the previous section is given in [4]. In this paper, for lack of space, we only report syntax and reduction rules, to give the reader a precise, yet rather intuitive, definition of the behaviour of the operators. However, in the real scenario (see Section 4) a component expression is not reduced at the source level, but rather generates a binary component in a context where binary components for component names used inside the expression are already available. This is modeled by the type system given in [4].

The framework is parametric, in the sense that syntax and reduction rules can be instantiated on top of a programming language providing some syntactic categories and judgments. We use a Java-oriented terminology, since our aim is to instantiate the framework on Java-like languages (in particular, in [4] we present an instantiation on Featherweight Java [15]). However, the framework could in principle be applied more in general, thinking of "class" as "language entity".

In order to define syntax and reduction semantics of our component language, we first list the syntactic categories the used programming languages must provide.

- Simple class names (`c`). A qualified class name has the shape `M.c`, where `c` is a simple class name, and `M` is a component name. The meta-variable `n` ranges over both the sets of simple and qualified class names.
- (Source) class definitions (`cd`$^{\mathsf{s}}$). We assume that each source class definition introduces a simple class name `c` that can be extracted by a function *out*. Sequences of source class definitions $\mathsf{cd}_1^{\mathsf{s}} \ldots \mathsf{cd}_n^{\mathsf{s}}$ will also be denoted by `S`.

The syntax used for creating and composing components is given in Fig.1. We assume that order in sequences is immaterial and use a bar notation for sequences following the same conventions as in [15] (for instance, $\overline{\mathsf{c}}$ stands for $\mathsf{c}_1 \ldots \mathsf{c}_n$).

An application program corresponds to an executable application obtained by assembling together and deploying some components as specified in the environment `MDS`, and by providing a main expression e^{s} from which execution must start in the context of components `MDS`.

A component environment is a sequence of component declarations (possibly mutually dependent), each one associated with a distinct name.

A basic component `BM` is a sequence of class names (the deferred classes), followed by a sequence of class definitions. We assume that all class names (deferred or defined) introduced in `BM` are distinct.

12

$$\begin{array}{rll}
\texttt{P} ::= & (\texttt{MDS}, \texttt{e}^{\texttt{s}}) & \text{application program} \\
\texttt{MDS} ::= & \overline{\texttt{MD}} & \text{(source) component environment} \\
\texttt{MD} ::= & \texttt{M} = \texttt{ME} & \text{component declaration} \\
\texttt{ME} ::= & \texttt{M} \mid \texttt{BM} \mid \texttt{merge}(\texttt{ME}_1, \texttt{ME}_2) \mid \texttt{restrict}(\texttt{ME}, \texttt{c}) \mid \text{component expression} \\
& \texttt{rename}(\texttt{ME}, \texttt{c} \mapsto \texttt{c}') \mid \\
& \texttt{bind}(\texttt{ME}, \texttt{d} \mapsto \texttt{n}) \mid \texttt{unbind}(\texttt{ME}, \texttt{c} \mapsto \texttt{d}) \\
\texttt{BM} ::= & \{\overline{\texttt{c}}; \texttt{S}\} & \text{basic component}
\end{array}$$

where: component/class names declared in $\texttt{MDS}/\texttt{BM}$ are distinct;
$in(\texttt{S}) \subseteq \overline{\texttt{c}} \cup out(\texttt{S})$ in $\texttt{BM}$

Fig. 1. Syntax

Moreover, we assume that class definitions can only contain soft links to classes which are explicitly declared in $\texttt{BM}$, either in $\overline{\texttt{c}}$ or in $\texttt{S}$. If $\texttt{S} = \texttt{cd}_1^{\texttt{s}} \ldots \texttt{cd}_n^{\texttt{s}}$, then $out(\texttt{S}) = out(\texttt{cd}_1^{\texttt{s}}) \cup \ldots \cup out(\texttt{cd}_n^{\texttt{s}})$ denotes the set of all classes defined in $\texttt{S}$, whereas $in(\texttt{S})$, whose definition depends on the used language, is expected to denote the set of all soft links in $\texttt{S}$. Recall that a soft link to a class is any of its unqualified occurrences except those which introduce the declaration of either the class itself, or any of its constructors.

For instance, in `component M={class C{ C(){...} M.C m(C arg){...}}}` only the last occurrence of `C` is a soft link to `C`, whereas `M.C` is a hard link, that is, a link permanently anchored to the declaration of `C` inside `M`.

Defined class names are not associated permanently with a class definition in the component, but their definition can be changed later when composing the component with others. In other words, classes are all considered *virtual*.

Composition operators include `merge`, `restrict`, `rename`, `bind`, and `unbind`. The reduction relations over programs, component environments, declarations and expressions are defined by the rules in Figure 2. For simplicity, we use the same symbol for the reduction relations over the four different sets of terms, since such sets are mutually disjoint.

Values for component expressions are basic components $\texttt{BM}$, whereas a component declaration $\texttt{M} = \texttt{ME}$ is expected to reduce to a declaration of a basic component $\texttt{M} = \texttt{BM}$. Analogously, component environments are expected to reduce to environments of basic components $\overline{\texttt{M} = \texttt{BM}}$.

Rule (*prog*) corresponds to the intuition that the component environment of the program needs first to be reduced to a collection of declarations of basic components; then, the reduced component environment is closed by completing simple class names with their corresponding qualified version, and, finally, in the context of the class definitions extracted from the elaborated component environment, the reduction of $\texttt{e}^{\texttt{s}}$ can start (*prog2*) according to the reduction relation $\rightarrow_{core}$ at the level of the programming language.

The auxiliary functions *classes* and *close* are trivially defined by

$$(prog)\frac{\text{MDS} \rightarrow \text{MDS}'}{(\text{MDS}, \mathsf{e}^\mathsf{s}) \rightarrow (\text{MDS}', \mathsf{e}^\mathsf{s})}$$

$$(prog2)\frac{(\mathsf{S}, \mathsf{e}^\mathsf{s}) \rightarrow_{core} (\mathsf{S}, \mathsf{e}^{\mathsf{s}'})}{(\overline{\mathtt{M} = \mathtt{BM}}, \mathsf{e}^\mathsf{s}) \rightarrow (\overline{\mathtt{M} = \mathtt{BM}}, \mathsf{e}^{\mathsf{s}'})} \quad \mathsf{S} \equiv classes(close(\overline{\mathtt{M} = \mathtt{BM}}))$$

$$(mdecs)\frac{\text{MD} \rightarrow \text{MD}'}{\overline{\mathtt{M} = \mathtt{BM}}\ \text{MD MDS} \rightarrow \overline{\mathtt{M} = \mathtt{BM}}\ \text{MD}'\ \text{MDS}}$$

$$(mdecs2)\frac{}{\overline{\mathtt{M} = \mathtt{BM}}\ \text{MD MDS} \rightarrow \overline{\mathtt{M} = \mathtt{BM}}\ \text{MD}'\ \text{MDS}} \quad \begin{array}{l} \text{MD}' \equiv \text{MD}[\overline{\mathtt{BM}/\mathtt{M}}] \\ \text{MD}' \not\equiv \text{MD} \end{array}$$

$$(mdec)\frac{\text{ME} \rightarrow \text{ME}'}{\mathtt{M} = \text{ME} \rightarrow \mathtt{M} = \text{ME}'}$$

$$(merge)\frac{}{\mathtt{merge}(\{\overline{c}_1; \mathsf{S}_1\}, \{\overline{c}_2; \mathsf{S}_2\}) \rightarrow \{\overline{c}; \mathsf{S}_1\mathsf{S}_2\}} \quad \begin{array}{l} \overline{c} = \overline{c}_1\overline{c}_2 \setminus out(\mathsf{S}_1\mathsf{S}_2) \\ out(\mathsf{S}_1) \cap out(\mathsf{S}_2) = \emptyset \end{array}$$

$$(restrict)\frac{}{\mathtt{restrict}(\{\overline{c}; \mathsf{S}\,\mathsf{cd}^\mathsf{s}\}, \mathsf{c}) \rightarrow \{\overline{c}\,\mathsf{c}; \mathsf{S}\}} \quad out(\mathsf{cd}^\mathsf{s}) = \mathsf{c}$$

$$(rename)\frac{}{\mathtt{rename}(\{\overline{c}; \mathsf{S}\}, \mathsf{c} \mapsto \mathsf{c}') \rightarrow \{\overline{c}; \mathsf{S}\}[\mathsf{c}'/\mathsf{c}]} \quad \begin{array}{l} \mathsf{c} \in \overline{c} \cup out(\mathsf{S}) \\ \mathsf{c}' \not\in \overline{c} \cup out(\mathsf{S}) \end{array}$$

$$(bind)\frac{}{\mathtt{bind}(\{\overline{c}\,\mathsf{d}; \mathsf{S}\}, \mathsf{d} \mapsto \mathsf{n}) \rightarrow \{\overline{c}; \mathsf{S}[\mathsf{n}/\mathsf{d}]\}} \quad \mathsf{n} \text{ qualified or } \mathsf{n} \in out(\mathsf{S})$$

$$(unbind)\frac{}{\mathtt{unbind}(\{\overline{c}; \mathsf{S}\}, \mathsf{c} \mapsto \mathsf{d}) \rightarrow \{\overline{c}\,\mathsf{d}; \mathsf{S}[\mathsf{d}/in\ \mathsf{c}]\}} \quad \begin{array}{l} \mathsf{c} \in out(\mathsf{S}) \\ \mathsf{d} \not\in \overline{c} \cup out(\mathsf{S}) \end{array}$$

Fig. 2. Reduction rules

$$classes(\overline{\mathtt{M} = \{\overline{c}; \mathsf{S}\}}) = \overline{\mathsf{S}}$$
$$close(\overline{\mathtt{M} = \{\overline{c}; \mathsf{S}\}}) = \overline{\mathtt{M} = \{\overline{c}; close_{\mathtt{M}}(\mathsf{S})\}}$$

The definition of $close_{\mathtt{M}}$, though trivial (simple class names are qualified by $\mathtt{M}$), depends on the used language; the instantiation for FJ can be found in [4].

In a component environment, component declarations are sequentially processed from left to right. The leftmost declaration $\mathtt{MD}$ which is not fully reduced yet is selected, and, either a reduction step can be applied to $\mathtt{MD}$ (*mdecs*), or some name $\mathtt{M}_i$ of previously declared components can be substituted with the corresponding basic expression (*mdecs2*). Note that even though the two rules are not mutually exclusive, the reduction relation turns out to be confluent. The side condition $\mathtt{MD}' \not\equiv \mathtt{MD}$ avoids loops, whereas $\mathtt{MD}[\overline{\mathtt{BM}/\mathtt{M}}]$ denotes parallel substitution

of M_i with BM_i, for $i \in 1..n$, in MD. The inductive definition of such substitution is standard, except for the following case: $\{\overline{c}; S\}[\overline{BM/M}] = \{\overline{c}; S\}$. Substitution is not propagated inside components, since hard links are allowed to establish mutual dependencies between components.

Rule (*mdec*) is straightforward.

We denote by $S[c'/in\ c]$ the class definitions obtained from S by replacing every soft link to c by c'. Recall that references to c are all occurrences of c except those which either occur in qualified names, or introduce the declaration of either c, or one of its constructors.

Finally, $\overline{c}[c'/c]$ denotes the replacement of c with c' in $\overline{c}$, if present, and $S[c'/c]$ denotes the replacement of simple class name c (but not of qualified names of shape $M.c$) with c'. That is, $S[c'/c]$ differs from $S[c'/in\ c]$ since it also replaces declaring occurrences. Again, the precise definitions of $\_[\_/in\ \_]$ and $\_[\_/\_]$ depend on the core language.

The reduction relation for component expressions is defined as the compatible closure of the corresponding rules, since, for brevity, we have omitted the usual congruence rules. Even though it is not deterministic, the reduction relation is clearly confluent by orthogonality.

Merging two basic components (*merge*) corresponds to just putting together their class definitions ($S_1\ S_2$), provided that there are no conflicts ($out(S_1) \cap out(S_2) = \emptyset$), whereas the deferred classes are those of the two components which do not match with a defined class ($\overline{c}_1\overline{c}_2 \setminus out(S_1 S_2)$); note that deferred classes are shared.

The restrict operator (*restrict*) removes the definition of a class c in a basic component, and makes c a deferred class.

The rename operator (*rename*) performs a bijective renaming of a class c into c' in a basic component BM: c must be either a deferred or a defined class in BM, whereas c' must be new, that is, neither deferred nor defined in BM. Recall that qualified names are not affected by the substitution.

The bind operator (*bind*) replaces all soft links to a deferred class[8] with the name of a defined class of the same component or with a qualified class name. Conversely, the unbind operator (*unbind*) replaces all soft links to a defined class with a new deferred class.

As final remark, note that all the composition operators can be expressed as a combination of operators in (mixin) module calculi, such as CMS [6]. Indeed, **merge** (called *link* in [6]) and **restrict** are exactly the corresponding operators of the CMS version with virtual components, whereas **rename**, **bind** and **unbind** can all be obtained as special instances of the CMS *reduct* operator which allows independent renaming of input and output names (in **rename** names which are both input and output are renamed in the same way, and only bijective renamings are considered; in **bind** an input name is renamed to an output name; finally, in **unbind** an input name is renamed to a fresh name). Hence, the semantics of our component language could be equivalently given by translation into

---

[8] Note that all soft links to a deferred class are just all unqualified occurrences of c, hence $S[n/d]$ and $S[n/in\ d]$ coincide here.

CMS. However, we preferred here a direct semantics since it is more intuitive for most readers. Note also that `unbind` operator, which seems at a first sight to change the inner structure of a component, actually can safely be expressed by module operators which consider a component as a black box, relying on the CMS distinction between (external) names and (internal) variables which we have omitted here for simplicity: that is, only the input name is changed, whereas the variable used in internal code is kept. This model exactly reflects what happens at the implementation level.

4 Implementation

In this section we discuss how we have implemented a prototype compiler for the framework we have presented; it can be downloaded (along with its sources and some examples) at: `http://www.disi.unige.it/person/LagorioG/SmartJavaComp/` This compiler supports a small Java subset, which extends the language used in the instantiation of the framework described in [4]; in addition to some syntactic shortcuts it supports primitive types, assignments, implicit use of `this`, the literal `null`, `void` methods, constructor overload and basic statements. All examples shown in the paper can be tested.

Our prototype consists of two programs: the *compiler* and the *deployer*. The former generates `.bc` component binary files from `.sjc` component source files, and the latter assembles component binary files into standard `.jar` files. These resulting JAR files are directly executable on any JVM (Java Virtual Machine).

A `.sjc` file contains a single component declaration `MD` as in Fig.1, where the language used for writing class definitions is the small Java subset described above. A `.bc` file (a binary component) is (roughly) a collection of Java classes in polymorphic bytecode format, each one equipped with its constraints. A basic component is compiled by compiling in isolation any class definition, by implementing the type system for separate compilation defined in [2], extended to the considered language.

Component declarations where unbound component names appear only in qualified names can be compiled in total isolation. On the other hand, component declarations which depend on other components can be compiled only if these components are already available in binary form. In this case, our compiler acts also as a linker, that is, it generates a new `.bc` file by also using those binary files.

When components are compiled, type constraints are checked for consistency; unfortunately, some errors could be undetected as long as components remain open. Luckily, verification of constraints is complete in case of closed components [2].

Because binary components contain polymorphic bytecode, they cannot be directly loaded, much less executed, by a standard JVM. In order to obtain a standard Java "executable" (that is, a JAR archive containing a proper *manifest*) from a set of `.bc` binary files, we must *deploy* them.

The deployer can assemble components into a single executable, after having checked that these components complete each other without clashing; that is, when the collection of Java class signatures extracted from these components is well-formed[9] and all type constraints of components can be simplified in this environment of class signatures.

5 Conclusion

We have presented a parametric framework of components for Java-like languages where a component is a collection of (binary) classes, each one equipped with type constraints on used classes. These type constraints guarantee *safe* linking of components; moreover, linking is *flexible*, in the sense that type constraints are abstract enough to never reject safe compositions, and components can be combined by a set of powerful (mixin) module operators.

A concrete instantiation of the framework can be provided by giving a suitable intermediate language: Java bytecode or .NET intermediate language does not allow fully adaptive components since, roughly speaking, they do not abstract away from all the possible contexts where open components can be safely used. However, as shown in [2], it is possible to define more abstract binary languages which are adequate to this aim. Our work until now, both in [2] and in the prototype accompanying this paper, has focused on extending Java bytecode, by adding type variables and type constraints. However, instantiations based on .NET intermediate language are feasible and interesting as well; moreover, they would be even more appealing, being .NET an intermediate language which does not rely on a particular source language, so the corresponding component framework would allow interoperability among components written in any language which targets .NET. We plan to investigate this possibility further.

Basic components are constructed, as mentioned above, in a particular language. Again, the framework can be instantiated on any source language which allows compilation in isolation of classes in the given binary language.

The semantics of the component language is defined in terms of reduction into basic components, that is, collection of class declarations.

To show the effectiveness of the approach, we have provided in [4] a complete formal description of an instantiation of the framework on Featherweight Java [15], which uses the type system for compositional compilation in [2]. Moreover, we have developed a prototype implementation on a small Java subset, which implements a large extension of this type system.

In literature there exist several proposals to better support component programming in object-oriented languages.

MzScheme [13] and Jiazzi [17] components are mixins which can be statically linked, in a way similar to our approach. MzScheme is built on top of Scheme and is not statically typed; Jiazzi is inspired by MzScheme, but is defined on top of Java, and is statically typed.

---

[9] The class hierarchy is acyclic and there is no bad overriding/overloading.

Other related papers propose language level abstractions for component-oriented programming allowing components to be first-class entities. ComponentJ [18], ArchJava [1], and ACOEL [19] are Java-like component-oriented languages, where components can be dynamically composed by explicitly connecting their *ports*. Ports basically play the role of required and provided interfaces in our framework.

ComponentJ promotes black-box object-oriented component programming style, by avoiding inheritance in favor of object composition.

ArchJava is an extension of Java with component classes; its type system allows for static checking of structural conformance between architecture and implementation.

ACOEL is an extensional language for supporting black-box components which uses mixins and virtual types to build adaptable applications.

Finally, Zenger [22] follows a more scalable approach, by proposing a component model where components are composed by type-safe high-level composition operators.

Differently to our approach, all the works above are less focused on the problem of programming language independence and interoperability of binary components. There are several short term enhancements on the design of the component language which could be considered: for instance, adding the possibility of hiding classes in components by making them private, or allowing non virtual classes (classes statically bound).

Long term future work includes at least two important directions. First, our binary components are linkable units, but not loadable units, that is, they cannot be replaced or serviced after application execution has started. Hence, we plan to study the possibility of considering a different semantics for the composition operators based on dynamic rather static linking, following the approach taken in [11, 16] where models for virtual machines able to execute polymorphic bytecode have been defined.

Another limitation of the approach is that mutual consistency of components only means that type correctness is guaranteed, but of course does not imply that components satisfy some expected behaviour. To go more towards preservation of also semantic properties, one should develop an assertion-based version of both required and provided interfaces.

References

1. J. Aldrich, C. Chambers, and D. Notkin. Architectural reasoning in ArchJava. In B. Magnusson, editor, *ECOOP'02 - Object-Oriented Programming*, number 2374 in Lecture Notes in Computer Science, pages 334–367. Springer, 2002.
2. D. Ancona, F. Damiani, S. Drossopoulou, and E. Zucca. Polymorphic bytecode: Compositional compilation for Java-like languages. In *ACM Symp. on Principles of Programming Languages 2005*. ACM Press, January 2005.
3. D. Ancona, G. Lagorio, and E. Zucca. Smart modules for Java-like languages. In *7th Intl. Workshop on Formal Techniques for Java-like Programs 2005*, July 2005.

4. D. Ancona, G. Lagorio, and E. Zucca. A flexible and type-safe framework of components for Java-like languages. Technical report, Dipartimento di Informatica e Scienze dell'Informazione, Università di Genova, 2006. Submitted for journal publication.

5. D. Ancona and E. Zucca. True modules for Java-like languages. In J.L. Knudsen, editor, *ECOOP'01 - European Conference on Object-Oriented Programming*, number 2072 in Lecture Notes in Computer Science, pages 354–380. Springer, 2001.

6. D. Ancona and E. Zucca. A calculus of module systems. *Journ. of Functional Programming*, 12(2):91–132, 2002.

7. G. Bracha. *The Programming Language JIGSAW: Mixins, Modularity and Multiple Inheritance.* PhD thesis, Department of Comp. Sci., Univ. of Utah, 1992.

8. G. Bracha, M. Odersky, D. Stoutmire, and P. Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *ACM Symp. on Object-Oriented Programming: Systems, Languages and Applications 1998*, pages 183–200. ACM Press, October 1998.

9. K. B. Bruce and J. N. Foster. LOOJ: Weaving LOOM into Java. In *ECOOP'04 - Object-Oriented Programming*, number 3086 in Lecture Notes in Computer Science, pages 389–413, 2004.

10. K.B. Bruce, M. Odersky, and P. Wadler. A statically safe alternative to virtual types. In *ECOOP'98 - European Conference on Object-Oriented Programming*, number 1445 in Lecture Notes in Computer Science, pages 523–549, 1998.

11. Alex Buckley and Sophia Drossopoulou. Flexible Dynamic Linking. In *6th Intl. Workshop on Formal Techniques for Java Programs 2004*, June 2004.

12. Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications.* Addison-Wesley, 2000.

13. R.B. Findler and M. Flatt. Modular object-oriented programming with units and mixins. In *Intl. Conf. on Functional Programming 1998*, September 1998.

14. T. Hirschowitz and X. Leroy. Mixin modules in a call-by-value setting. In D. Le Métayer, editor, *ESOP 2002 - European Symposium on Programming 2002*, number 2305 in Lecture Notes in Computer Science, pages 6–20. Springer, 2002.

15. A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.

16. G. Lagorio. Dynamic linking of polymorphic bytecode. In *8th Intl. Workshop on Formal Techniques for Java-like Programs 2005*, July 2006.

17. S. McDirmid, M.Flatt, and W. Hsieh. Jiazzi: New age components for old fashioned Java. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 2001)*. ACM Press, October 2001.

18. J. Costa Seco and L. Caires. A basic model of typed components. In E. Bertino, editor, *ECOOP'00 - European Conference on Object-Oriented Programming*, number 1850 in Lecture Notes in Computer Science, pages 108–128. Springer, 2000.

19. V. C. Sreedhar. Mixin'up components. In *Proceedings of the 22rd International Conference on Software Engineering, ICSE 2002*, pages 198–207. ACM, 2002.

20. Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming, 2nd Edition.* Addison Wesley, 2002.

21. M. Torgersen. The expression problem revisited. In M. Odersky, editor, *ECOOP'04 - Object-Oriented Programming*, number 3086 in Lecture Notes in Computer Science, pages 123–143. Springer, 2004.

22. M. Zenger. Type-safe prototype-based component evolution. In *ECOOP'02 - Object-Oriented Programming*, number 2374 in Lecture Notes in Computer Science, pages 470–497, Berlin, 2002. Springer.