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 programming language; of course, basic components still need to be constructed by using some programming language. For instance, .NET assemblies do not strongly rely on any particular language, but can be created, for instance, both from C# and Haskell code. However assembling components is a process

⋆ This work has been partially supported by APPSEM II - Thematic network IST-2001-38957, and MIUR EOS - Extensible Object Systems.

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 [8, 5, 22, 16, 7, 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 logically 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 [6]. 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

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

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 [8, 7], 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 the framework, by listing the ingredients the underlying Java-like language should provide. We give reduction semantics and typing rules, and show soundness of the type system. 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]. An extended version of this paper can be found in [4]; it includes more examples and the formal description of an instantiation of the framework on top of Featherweight Java [15].

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;}
  }
}
```

[2] Which, however, can be encoded in lower-level operators of module calculi such as CMS [7].
[3] `http://www.disi.unige.it/person/LagorioG/SmartJavaComp/`
[4] For simplicity, we will keep the examples small and avoid access modifiers.

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 shown below, 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* 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 definition 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, the required interface should specify, on the one

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

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; 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 [6], is too restrictive in the other respect, 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 List@LinkedList(); // type error
```

The qualified name `List@LinkedList` 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*.[7]

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

[6] To avoid ambiguities with the syntax used by Java-like languages for member accesses, we prefer to avoid the dot notation for components.

[7] See more in 2.5.

Bind A closed component can be obtained by binding the deferred classes of some open component to declarations 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;}
  }
}
```

When closing a component, all type constraints in the class types must be verified, otherwise a type error is issued. 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[8] 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.

[8] 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.
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.
The restrict operator provides another mean for opening closed components. It is mainly used jointly with the merge operator to override class declarations. 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 declaration. Then the new declaration 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 declaration of `C`, but redirects soft links to `C` to an unused class `U`; `restrict(Comp,C)`, instead, makes class `C` deferred by removing its declaration, 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 { Node@AComponent(Node@AComponent); }
    Node@AComponent first;
    void addFirst(){first=new Node@AComponent(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 which can always be redirected by the composition operators, hard links cannot be redirected[9] 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(Node@YetAnotherList); }
    Node@YetAnotherList first;
    void addFirst(){first=new Node@YetAnotherList(first);}
  }
  class Node{
    requires { & Node@YetAnotherList; }
    Node@YetAnotherList next;
    Node(Node@YetAnotherList n){next=n;}
    Node@YetAnotherList 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.
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->Node@YetAnotherList);
```

3 A Framework of Components

In this section, we define a parametric framework for components which 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" and of "binary" as abstract intermediate language.

[9] However, the motivation for this limitation is methodological rather than technical.

3.1 Syntax and reduction rules

In order to define syntax and reduction semantics of our component language, we first list the syntactic categories the used programming languages must provide; the list of required judgments will be given when describing the type system.

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

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

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

Fig. 1. Syntax

An application program corresponds to an executable application obtained by assembling together and deploying some components as specified in the environment $\mathtt{MDS}$, and by providing a main expression $\mathtt{e}^{\mathsf{s}}$ from which execution must start in the context of components $\mathtt{MDS}$.
A component environment is a sequence of component declarations (possibly mutually dependent), each one associated with a distinct name.
A basic component $\mathtt{BM}$ 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 $\mathtt{BM}$ are distinct.
Moreover, we assume that class definitions can only contain soft links to classes which are explicitly declared in $\mathtt{BM}$, either in $\overline{\mathtt{c}}$ or in S. If $\mathsf{S} = \mathsf{cd}^{\mathsf{s}}{}_1 \ldots \mathsf{cd}^{\mathsf{s}}{}_n$, then $out(\mathsf{S}) = out(\mathsf{cd}^{\mathsf{s}}{}_1) \cup \ldots \cup out(\mathsf{cd}^{\mathsf{s}}{}_n)$ denotes the set of all classes defined in S, whereas $in(\mathsf{S})$, whose definition depends on the used language, is expected to denote the set of all soft links in 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. Besides simple class names, a class definition can contain qualified class names, that is, hard links to classes defined in other components.
For instance, in `component M={class C{ C(){...} C@M m(C c){...}}}` only the last occurrence of `C` is a soft link to `C`, whereas `C@M` is a hard link, that is, a link permanently anchored to the declaration of `C` inside `M`.

Note that 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 in components are all implicitly 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 defined in Figure 2. For simplicity, we use the same symbol for the reduction relations over the four different set of terms, since such terms are mutually disjoint.
Values for component expressions are basic components $\mathtt{BM}$, whereas a component declaration $\mathtt{M} = \mathtt{ME}$ is expected to reduce to a declaration of a basic component $\mathtt{M} = \mathtt{BM}$. Analogously, component environments are expected to reduce to environments of basic components.
Note that the reduction semantics is provided only to be able to express soundness of composition of components (formally, a component environment $\mathtt{MDS}$) w.r.t. global compilation of the corresponding classes, that is, the collection of classes which we get by reducing and then deploying (see below) $\mathtt{MDS}$ (Theorem 3 at the end of this Section). This allows to prove the soundness of component composition in a modular way, that is, by relying on type soundness of the used programming language. 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 are already available. This is modeled by the type system in the following.
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 $\mathtt{e}^{\mathtt{s}}$ 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

$$\begin{aligned} \mathit{classes}(\overline{\mathtt{M} = \{\overline{\mathtt{c}}; \mathtt{S}\}}) &= \overline{\mathtt{S}} \\ \mathit{close}(\overline{\mathtt{M} = \{\overline{\mathtt{c}}; \mathtt{S}\}}) &= \overline{\mathtt{M} = \{\overline{\mathtt{c}}; \mathit{close}_{\mathtt{M}}(\mathtt{S})\}} \end{aligned}$$

The definition of $\mathit{close}_{\mathtt{M}}$, though trivial as well (simple class names are qualified by $\mathtt{M}$), depends on the used language; the instantiation for Featherweight Java can 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 $\mathtt{M}_i$ with $\mathtt{BM}_i$, for $i \in 1..n$, in $\mathtt{MD}$. The inductive definition of such substitution is standard, except for the following case:

$$\{\overline{\mathtt{c}}; \mathtt{S}\}[\overline{\mathtt{BM}/\mathtt{M}}] = \{\overline{\mathtt{c}}; \mathtt{S}\}.$$

Substitution is not propagated inside components, since hard links are allowed to establish mutual dependencies between components.
Rule (*mdec*) is straightforward.
We denote by $\mathtt{S}[\mathtt{c}'/\mathit{in}\ \mathtt{c}]$ the class definitions obtained from $\mathtt{S}$ by replacing every soft link to $\mathtt{c}$ by $\mathtt{c}'$. Recall that references to $\mathtt{c}$ are all occurrences of $\mathtt{c}$ except those which either occur in qualified names, or introduce the declaration of either $\mathtt{c}$, or one of its constructors.

$$\textit{(prog)}\;\frac{\mathtt{MDS} \longrightarrow \mathtt{MDS}'}{(\mathtt{MDS}, \mathsf{e}^{\mathsf{s}}) \longrightarrow (\mathtt{MDS}', \mathsf{e}^{\mathsf{s}})}$$

$$\textit{(prog2)}\;\frac{(\mathsf{S}, \mathsf{e}^{\mathsf{s}}) \longrightarrow_{core} (\mathsf{S}, \mathsf{e}^{\mathsf{s}\prime})}{(\overline{\mathtt{M} = \mathtt{BM}}, \mathsf{e}^{\mathsf{s}}) \longrightarrow (\overline{\mathtt{M} = \mathtt{BM}}, \mathsf{e}^{\mathsf{s}\prime})}\quad \mathsf{S} \equiv \mathit{classes}(\mathit{close}(\overline{\mathtt{M} = \mathtt{BM}}))$$

$$\textit{(mdecs)}\;\frac{\mathtt{MD} \longrightarrow \mathtt{MD}'}{\overline{\mathtt{M} = \mathtt{BM}}\;\mathtt{MD}\;\mathtt{MDS} \longrightarrow \overline{\mathtt{M} = \mathtt{BM}}\;\mathtt{MD}'\;\mathtt{MDS}}$$

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

$$\textit{(mdec)}\;\frac{\mathtt{ME} \longrightarrow \mathtt{ME}'}{\mathtt{M} = \mathtt{ME} \longrightarrow \mathtt{M} = \mathtt{ME}'}$$

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

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

$$\textit{(rename)}\;\frac{}{\mathtt{rename}(\{\overline{\mathtt{c}}; \mathsf{S}\}, \mathtt{c} \mapsto \mathtt{c}') \longrightarrow \{\overline{\mathtt{c}}; \mathsf{S}\}[\mathtt{c}'/\mathtt{c}]}\quad \begin{array}{l}\mathtt{c} \in \overline{\mathtt{c}} \cup \mathit{out}(\mathsf{S})\\ \mathtt{c}' \notin \overline{\mathtt{c}} \cup \mathit{out}(\mathsf{S})\end{array}$$

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

$$\textit{(unbind)}\;\frac{}{\mathtt{unbind}(\{\overline{\mathtt{c}}; \mathsf{S}\}, \mathtt{c} \mapsto \mathtt{d}) \longrightarrow \{\overline{\mathtt{c}}\,\mathtt{d}; \mathsf{S}[\mathtt{d}/\mathit{in}\ \mathtt{c}]\}}\quad \begin{array}{l}\mathtt{c} \in \mathit{out}(\mathsf{S})\\ \mathtt{d} \notin \overline{\mathtt{c}} \cup \mathit{out}(\mathsf{S})\end{array}$$

Fig. 2. Reduction rules

Finally, $\overline{\mathtt{c}}[\mathtt{c}'/\mathtt{c}]$ denotes the replacement of $\mathtt{c}$ with $\mathtt{c}'$ in $\overline{\mathtt{c}}$, if present, and $\mathsf{S}[\mathtt{c}'/\mathtt{c}]$ denotes the replacement of simple class name $\mathtt{c}$ (but not of qualified names of shape $\mathtt{c@M}$) with $\mathtt{c}'$. That is, $\overline{\mathtt{c}}[\mathtt{c}'/\mathtt{c}]$ differs from $\mathsf{S}[\mathtt{c}'/\mathit{in}\ \mathtt{c}]$ since it also replaces declaring occurrences. Again, the precise definitions of $\_[\_/\mathit{in}\ \_]$ and $\_[\_/\_]$ depend on the used 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 ($\mathsf{S}_1\,\mathsf{S}_2$), provided that there are no conflicts, whereas the deferred classes are those of the two components which do not match with a defined class ($\overline{\mathtt{c}}_1\overline{\mathtt{c}}_2 \setminus \mathit{out}(\mathsf{S}_1\mathsf{S}_2)$); note that deferred classes are shared.

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

The rename operator (*rename*) performs a bijective renaming of a class $\mathtt{c}$ into $\mathtt{c}'$ in a basic component $\mathtt{BM}$: $\mathtt{c}$ must be either a deferred or a defined class in $\mathtt{BM}$, whereas $\mathtt{c}'$ 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[10] 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 [7]. Indeed, `merge` (called *link* in [7]) 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 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.

3.2 Type system

We describe now types and typing rules for our component language. First, we list the additional syntactic categories and judgments the used programming language must provide.

- Binary class definitions (cd^{b}). We assume that each binary class definition introduces a simple class name c that can be extracted by a function *out*. Sequences of binary class definitions ${\mathsf{cd}^{\mathsf{b}}}_1 \ldots {\mathsf{cd}^{\mathsf{b}}}_n$ will be also denoted by B.
- Class signatures (δ), which can be thought of as the type information which can be extracted from a class definition (the class definition deprived of body). Function *out* is defined on class signatures as well. Sequences of class signatures are also denoted by Δ.
- Global compilation $\vdash_{core} \mathsf{S} : \Delta|\mathsf{B}$, to be read: the program (sequence of class definitions) S has class signatures Δ and compiles to the sequence of binary class definitions B.
- Type constraints (γ), which express requirements needed by a class for its correct functioning, e.g. that a given class has a field of a given type. Sequences of type constraints will be denoted also by Γ.
- Compositional compilation (of a class), $\vdash_{core} \mathsf{cd}^{\mathsf{s}} : \Gamma|\delta|\mathsf{cd}^{\mathsf{b}}$, to be read: The class definition cd^{s} has signature δ and compiles to cd^{b} under the type constraints in Γ.
- Linking, $\Delta \vdash_{core} \Gamma|\mathsf{cd}^{\mathsf{b}} \rightsquigarrow \Gamma'|\mathsf{cd}^{\mathsf{b}'}$, to be read: In the class signatures Δ the type constraints Γ are consistent and can be simplified into Γ', and the binary cd^{b} becomes $\mathsf{cd}^{\mathsf{b}'}$.

Types for our component language are given in Fig.3. Note that the type system for the component language models not only typechecking of component declarations, but, even more importantly, how these component declarations generate new binary components via compilation of defined classes and, possibly, linking of binary components already present. In other words, the type system models the semantics of our component framework at the binary level, as it should be implemented, and indeed is in the prototype we have developed. As a consequence, types play also the role of binaries, as we stress by the double terminology in the figure below.

[10] Note that all soft links to a deferred class are just all unqualified occurrences of `c`.

$\mathcal{M} ::= (\mathtt{M}_1, \mathtt{MT}_1) \ldots (\mathtt{M}_n, \mathtt{MT}_n)$	component type environment (binary component environment)
$\mathtt{MT} ::= \{\overline{\mathtt{c}}; \overline{\mathtt{CT}}\}$	component type (binary component)
$\mathtt{CT} ::= \Gamma \vert \delta \vert \mathsf{cd}^{\mathsf{b}}$	class type (binary class)

Fig. 3. Types

A *component type environment* is a sequence of pairs consisting of a component name and a component type, where all component names are assumed to be distinct. A *component type* contains the information needed to safely use a component in a context, and consists of a sequence of deferred classes and a sequence of *class types*. A class type models the binary code corresponding to a class, such as a `.class` file in Java; however, here this binary code should allow to extract, besides the class signature (the provided interface of the class), also the constraints (the required interface).

Typing rules are given in Fig.4. The expression *in*(CT), whose definition depends on the used language, is expected, analogously to *in*(cd^{s}), to denote the set of all soft links in $\mathtt{CT} = \Gamma \vert \delta \vert \mathsf{cd}^{\mathsf{b}}$ (that is, all class names in Γ and all simple class names in δ, cd^{b}, except those which introduce the declaration of a class, or any of its constructors). Also analogously to what we have done for the source component language, we assume function *out* to be naturally extended to sequences and to (sequences of) class types, and the used language to provide substitutions $\_[\_/in\ \_]$ and $\_[\_/\_]$ on class types..

A program (*prog*) is well typed if its component environment has a component type environment $\mathcal{M}$ that can be turned into a well formed *closed* component type environment $\mathcal{M}'$. If so, then the class signatures Δ extracted from $\mathcal{M}'$ are used for typing the main expression e^{s}. The judgment $\Delta \vdash_{core} \mathsf{e}^{\mathsf{s}} : \mathtt{c}$ depends on the used language.

All auxiliary rules and functions needed for (*prog*) are defined in Figure 5. Function *classBin* is not directly used in the typing rules, but is needed for stating the soundness result (see Theorem 3 below). When closing a component type environment (first rule), all simple class names appearing in the component types are qualified by the corresponding component name, as happens in the reduction rule for programs. Indeed, the functions *close* and $close_{\mathtt{M}}$ are the static counterpart of the (deliberately overloaded) functions used in the dynamic semantics. Then it must be checked that the resulting types are well formed closed component types w.r.t. the class signatures extracted from all component types. This means (second rule) that all constraints must be satisfied by the class signatures, that is, they all simplify to the empty set of constraints. Rules (*mdecs*) and (*mdec*) are standard.

Rule (*mtype*) defines well formed, but not necessarily closed, component types. Indeed, it checks whether all constraints of a component type MT are consistent w.r.t. the class signatures extracted from MT; therefore it is sufficient that the set of constraints provably simplifies to some other set of constraints (possibly the same). The notation $\Delta \vdash \Gamma \vert \mathsf{cd}^{\mathsf{b}} \diamond$ is an abbreviation for $\Delta \vdash_{core} \Gamma \vert \mathsf{cd}^{\mathsf{b}} \leadsto \Gamma' \vert \mathsf{cd}^{\mathsf{b}\prime}$ for some $\Gamma', \mathsf{cd}^{\mathsf{b}\prime}$. Note that this judgment is not strictly necessary for ensuring the soundness of the type system (indeed, mutual consistency of all components is checked again in rule (prog)), nevertheless it is used in rules (*basic*), (*merge*), and (*bind*) for guaranteeing earlier error detection.

In rule (*basic*), the type of a basic component is inferred by separately typechecking each class definition, obtaining the constraints on the used classes, the signature of the declared class, and the corresponding binary class. Recall that the formal definition of the judgment $\vdash_{core} \mathsf{cd}^{\mathsf{s}} : \Gamma \vert \delta \vert \mathsf{cd}^{\mathsf{b}}$ depends on the used programming language. Then, the obtained component

$$\textit{(prog)}\ \frac{\vdash \mathtt{MDS} : \mathcal{M} \qquad \vdash \mathcal{M} \leadsto_{close} \mathcal{M}' \qquad \Delta \vdash_{core} \mathsf{e}^{\mathsf{s}} : \mathtt{c}}{\vdash (\mathtt{MDS}, \mathsf{e}^{\mathsf{s}})\diamond}\ \Delta \equiv \mathit{classTypes}(\mathcal{M}')$$

$$\textit{(mdecs)}\ \frac{\emptyset \vdash \mathtt{MD}_1 : (\mathtt{M}_1, \mathtt{MT}_1) \ \cdots (\mathtt{M}_i, \mathtt{MT}_i)^{i \in 1..n-1} \vdash \mathtt{MD}_n : (\mathtt{M}_n, \mathtt{MT}_n)}{\vdash \overline{\mathtt{MD}} : \overline{(\mathtt{M}, \mathtt{MT})}}$$

$$\textit{(mdec)}\ \frac{\mathcal{M} \vdash \mathtt{ME} : \mathtt{MT}}{\mathcal{M} \vdash \mathtt{M} = \mathtt{ME} : (\mathtt{M}, \mathtt{MT})}$$

$$\textit{(mtype)}\ \frac{\overline{\delta} \vdash \Gamma_i | \mathsf{cd}_i^{\mathsf{b}} \diamond \ \ \forall\, i \in 1..n}{\vdash \{\overline{\mathtt{c}}; \overline{\Gamma | \delta | \mathsf{cd}^{\mathsf{b}}}\}\diamond}\ \ \mathit{in}(\Gamma_i | \delta_i | \mathsf{cd}_i^{\mathsf{b}}) \subseteq \overline{\mathtt{c}} \cup \mathit{out}(\overline{\delta}) \ \forall\, i \in 1..n$$

$$\textit{(mname)}\ \frac{}{\mathcal{M} \vdash \mathtt{M} : \mathtt{MT}}\ (\mathtt{M}, \mathtt{MT}) \in \mathcal{M}$$

$$\textit{(basic)}\ \frac{\vdash_{core} \mathsf{cd}_i^{\mathsf{s}} : \Gamma_i | \delta_i | \mathsf{cd}_i^{\mathsf{b}} \ \forall i \in 1..n \qquad \vdash \{\overline{\mathtt{c}}; \overline{\Gamma | \delta | \mathsf{cd}^{\mathsf{b}}}\}\diamond}{\mathcal{M} \vdash \{\overline{\mathtt{c}}; \overline{\mathsf{cd}^{\mathsf{s}}}\} : \{\overline{\mathtt{c}}; \overline{\Gamma | \delta | \mathsf{cd}^{\mathsf{b}}}\}}$$

$$\textit{(merge)}\ \frac{\mathcal{M} \vdash \mathtt{ME}_i : \{\overline{\mathtt{c}}_i; \overline{\mathtt{CT}}_i\}, i = 1,2 \qquad \vdash \{\overline{\mathtt{c}}; \overline{\mathtt{CT}}_1 \overline{\mathtt{CT}}_2\}\diamond}{\mathcal{M} \vdash \mathtt{merge}(\mathtt{ME}_1, \mathtt{ME}_2) : \{\overline{\mathtt{c}}; \overline{\mathtt{CT}}_1 \overline{\mathtt{CT}}_2\}}\ \begin{array}{l} \overline{\mathtt{c}} = \overline{\mathtt{c}_1}\,\overline{\mathtt{c}_2} \setminus \mathit{out}(\overline{\mathtt{CT}}_1 \overline{\mathtt{CT}}_2) \\ \mathit{out}(\overline{\mathtt{CT}}_1) \cap \mathit{out}(\overline{\mathtt{CT}}_2) = \emptyset \end{array}$$

$$\textit{(restrict)}\ \frac{\mathcal{M} \vdash \mathtt{ME} : \{\overline{\mathtt{c}}; \overline{\mathtt{CT}}\, \Gamma | \delta | \mathsf{cd}^{\mathsf{b}}\}}{\mathcal{M} \vdash \mathtt{restrict}(\mathtt{ME}, \mathtt{c}) : \{\overline{\mathtt{c}}\, \mathtt{c}; \overline{\mathtt{CT}}\}}\ \mathit{out}(\delta) = \mathtt{c}$$

$$\textit{(rename)}\ \frac{\mathcal{M} \vdash \mathtt{ME} : \{\overline{\mathtt{c}}; \overline{\mathtt{CT}}\}}{\mathcal{M} \vdash \mathtt{rename}(\mathtt{ME}, \mathtt{c} \mapsto \mathtt{c}') : \{\overline{\mathtt{c}}; \overline{\mathtt{CT}}\}[\mathtt{c}'/\mathtt{c}]}\ \begin{array}{l} \mathtt{c} \in \overline{\mathtt{c}} \cup \mathit{out}(\overline{\mathtt{CT}}) \\ \mathtt{c}' \notin \overline{\mathtt{c}} \cup \mathit{out}(\overline{\mathtt{CT}}) \end{array}$$

$$\textit{(bind)}\ \frac{\mathcal{M} \vdash \mathtt{ME} : \{\overline{\mathtt{c}}\, \mathtt{d}; \overline{\mathtt{CT}}\} \qquad \vdash \{\overline{\mathtt{c}}; \overline{\mathtt{CT}}[\mathtt{n}/\mathtt{d}]\}\diamond}{\mathcal{M} \vdash \mathtt{bind}(\mathtt{ME}, \mathtt{d} \mapsto \mathtt{n}) : \{\overline{\mathtt{c}}; \overline{\mathtt{CT}}[\mathtt{n}/\mathtt{d}]\}}\ \mathtt{n} \text{ qualified or } \mathtt{n} \in \mathit{out}(\overline{\mathtt{CT}})$$

$$\textit{(unbind)}\ \frac{\mathcal{M} \vdash \mathtt{ME} : \{\overline{\mathtt{c}}; \overline{\mathtt{CT}}\}}{\mathcal{M} \vdash \mathtt{unbind}(\mathtt{ME}, \mathtt{c} \mapsto \mathtt{d}) : \{\overline{\mathtt{c}}\, \mathtt{d}; \overline{\mathtt{CT}}[\mathtt{d}/\mathit{in}\ \mathtt{c}]\}}\ \begin{array}{l} \mathtt{c} \in \mathit{out}(\overline{\mathtt{CT}}) \\ \mathtt{d} \ \notin \overline{\mathtt{c}} \cup \mathit{out}(\overline{\mathtt{CT}}) \end{array}$$

Fig. 4. Typing rules

type must be checked (judgment $\vdash \{\overline{\mathtt{c}}; \overline{\Gamma | \delta | \mathsf{cd}^{\mathsf{b}}}\}\diamond$) in order to detect internal inconsistencies that would prevent the component to be effectively usable in any program.

In rule (*merge*), the operator can be safely applied only if the arguments have no conflicting class definitions. As in (*basic*), the resulting component type is required to be well formed, since in the merged components some constraint of a component operand could be inconsistent w.r.t. some of the class declared in the other operand.

In rule (*restrict*), the operator can be safely applied only if the class to be removed is actually defined in the component. The resulting component type is then obtained by removing the corresponding class type from the component type of the argument and by adding the class to the sequence of deferred classes. In this case, no checking on the resulting component type is needed, since the operation cannot introduce any sort of inconsistencies.

In rule (*rename*), the operator can be safely applied only if the class to be renamed is either a defined or a deferred class of the component, and the new name does not coincide with any

$$\frac{classTypes(\overline{\mathtt{MT}'}) \vdash \mathtt{MT}'_i \diamond_{closed} \;\; \forall i \in 1..n}{\vdash \overline{(\mathtt{M},\mathtt{MT})} \leadsto_{close} \overline{(\mathtt{M},\mathtt{MT}')}} \;\; \mathtt{MT}'_i = close_{\mathtt{M}_i}(\mathtt{MT}_i)\ \forall\, i \in 1..n$$

$$\frac{\varDelta \vdash_{core} \varGamma_i|\mathsf{cd}^{\mathsf{b}}_{\mathsf{i}} \leadsto \emptyset|\mathsf{cd}^{\mathsf{b}\prime}_{\mathsf{i}} \;\; \forall i \in 1..n}{\varDelta \vdash \{\emptyset; \overline{\varGamma|\delta|\mathsf{cd}^{\mathsf{b}}}\}\diamond_{closed}}$$

$$\begin{aligned}
classTypes(\overline{(\mathtt{M},\mathtt{MT})}) &= \overline{classTypes(\mathtt{MT})}\\
classTypes(\{\overline{\mathsf{c}}; \overline{\varGamma|\delta|\mathsf{cd}^{\mathsf{b}}}\}) &= \overline{\delta}
\end{aligned}$$

$$\begin{aligned}
classBin(\overline{(\mathtt{M},\mathtt{MT})}) &= \overline{classBin(\mathtt{MT})}\\
classBin(\{\overline{\mathsf{c}}; \overline{\varGamma|\delta|\mathsf{cd}^{\mathsf{b}}}\}) &= \overline{\mathsf{cd}^{\mathsf{b}}}
\end{aligned}$$

$$close_{\mathtt{M}}(\{\overline{\mathsf{c}}; \overline{\varGamma|\delta|\mathsf{cd}^{\mathsf{b}}}\}) = (\{\overline{\mathsf{c}}; \overline{close_{\mathtt{M}}(\varGamma)|close_{\mathtt{M}}(\delta)|close_{\mathtt{M}}(\mathsf{cd}^{\mathsf{b}})}\})$$

Fig. 5. Auxiliary rules and functions

class of the component. The resulting component type is obtained by correspondingly renaming the component type of the argument. Like it happens for (*restrict*), no checking on the resulting component type is needed.
In rule (*bind*), the operator can be safely applied only if `d` is deferred; if so, then `d` is bound to `n`. The resulting component type is obtained by replacing all soft links to `d` with `n` in the component type of the argument, and by removing `d` from the sequence of deferred classes. As in (*basic*) and (*merge*), the resulting component type is required to be well formed.
In rule (*unbind*), the operator can be safely applied only if `c` is a defined class and `d` does not coincide with any either defined or deferred class of the component. The resulting component type is obtained by replacing all soft links to `c` by `d` in the component type of the argument, and by adding `d` in the sequence of deferred classes. As for the restrict and the rename operator, no inconsistencies can be introduced, therefore the resulting component type does not need to be checked.
The relevance of the type system presented until now is that it supports *compositional compilation* of components. This means that it is possible for the programmer to write classes and compile them into a component in isolation, and then to compose the obtained component with other components by just checking that mutual assumptions are satisfied, without any need of re-inspecting code. This means that the framework truly supports components, that is, mixin modules in binary form.
Of course, we have to show that the compositional approach, where we first compile components in isolation and then combine them by checking their compatibility, is *sound*. That is, it gives the same result we would have obtained by not using components at all, but by compiling together all the classes obtained by reducing and then flattening components. Theorem 3 below states that this property holds under the assumption that compositional compilation of classes provided by the used language is sound w.r.t. global compilation.
The theorem can be proved by means of subject reduction and unique normal form properties stated in Theorem 1 and 2, respectively.
In the following, we assume that the used language satisfies the following assumption.

Assumption 1 (Soundness of compositional compilation of the used language) *If* $\vdash_{core} \mathsf{cd}^{\mathsf{s}}_{\mathsf{i}} : \varGamma_i|\delta_i|\mathsf{cd}^{\mathsf{b}}_{\mathsf{i}}$ *and* $\overline{\delta} \vdash_{core} \varGamma_i|\mathsf{cd}^{\mathsf{b}}_{\mathsf{i}} \leadsto \emptyset|\mathsf{cd}^{\mathsf{b}\prime}_{\mathsf{i}}$*, for* $i \in 1..n$

then $\vdash_{core} \overline{\mathsf{cd}^{\mathsf{s}}} : \overline{\delta}|\overline{\mathsf{cd}^{\mathsf{b}}}$.

Theorem 1 (Subject reduction). *If* $\vdash \mathtt{MDS} : \mathcal{M}$, $\mathtt{MDS} \rightarrow \mathtt{MDS'}$, *then* $\vdash \mathtt{MDS'} : \mathcal{M}$.

Theorem 2 (Unique normal form). *If* $\vdash \mathtt{MDS} : \mathcal{M}$, *then* $\mathtt{MDS} \xrightarrow{*} \mathtt{MDS'}$ *for a unique* $\mathtt{MDS'}$ *having shape* $\overline{\mathtt{M} = \mathtt{BM}}$.

Theorem 3. *If* $\vdash \mathtt{MDS} : \mathcal{M}$, *and* $\vdash \mathcal{M} \leadsto_{close} \mathcal{M}'$, *then* $\mathtt{MDS} \xrightarrow{*} \mathtt{MDS'}$ *for a unique* $\mathtt{MDS'} \equiv \overline{\mathtt{M} = \mathtt{BM}}$, *and* $\vdash_{core} classes(close(\mathtt{MDS'})) : \mathsf{B}|\Delta$ *with* $\mathsf{B} = classBin(\mathcal{M}')$ *and* $\Delta = classTypes(\mathcal{M}')$.

Theorem 3 states that, if by composing components we have obtained a new component which is a collection of class binary definitions and class signatures $\mathsf{B}|\Delta$, then such a component could be equivalently obtained from direct global compilation of the corresponding collection of classes (that is, those obtained by reducing the component expressions). This result implies as a corollary that composition of components is type safe, provided that the type system for the used language is sound.

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, which generates `.bc` component binary files from `.sjc` component source files, and
- the deployer, which 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) corresponds to a component type `MT` as in Fig.3, hence is (roughly) a collection of Java classes in polymorphic bytecode format, each one equipped with its constraints. The compiler implements typing rules in Fig.4. In particular, 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 (this corresponds to the $\mathcal{M}$ component type environment used in the typing rules). 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 (this corresponds to the step of closing a component type environment $\mathcal{M}$ in rule (prog) in Fig.4). 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 (class hierarchy is acyclic and there is no bad overriding/overloading);
– all type constraints of components can be simplified in this environment of class signatures.

These checks, as those made by the compiler for checking consistency of components, correspond to the judgment $\Delta \vdash_{core} \Gamma|\mathsf{B} \leadsto \Gamma'|\mathsf{B}'$, which is dependent on the used programming language (note that this judgment also includes checking well-formedness of Δ); our compiler and deployer implement the definition given in [4].

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 in the sense that, being .NET an intermediate language which does not rely on a particular source language, the corresponding component framework would allow interoperability among components written in any language which targets .NET. We plan to further investigate this possibility in further work.
Basic components are constructed, as mentioned above, in a particular programming language. Again, the framework can be instantiated on any source programming 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. The type system guarantees subject reduction and unique normal form for component expressions; moreover, composition of components is proved to be equivalent to global compilation of all their classes, hence to be type safe.
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 it is defined on top of Java, and is statically typed.

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 [23] 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 by Buckley and Drossopoulou [11] who have defined a model for a virtual machine able to execute polymorphic bytecode.
Second, 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 interfacesTo go more towards the preservation of semantic properties too, 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. Flexible type-safe linking of components for Java-like languages. Technical report, Dipartimento di Informatica e Scienze dell'Informazione, Università di Genova, March 2006.
5. D. Ancona and E. Zucca. A primitive calculus for module systems. In G. Nadathur, editor, *PPDP'99 - Principles and Practice of Declarative Programming*, number 1702 in Lecture Notes in Computer Science, pages 62–79. Springer, 1999.
6. 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.

7. D. Ancona and E. Zucca. A calculus of module systems. *Journ. of Functional Programming*, 12(2):91–132, 2002.
8. G. Bracha. *The Programming Language JIGSAW: Mixins, Modularity and Multiple Inheritance.* PhD thesis, Department of Comp. Sci., Univ. of Utah, 1992.
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. E. Machkasova and F.A. Turbak. A calculus for link-time compilation. In *ESOP 2000 - European Symposium on Programming 2000*, number 1782 in Lecture Notes in Computer Science, pages 260–274. Springer, 2000.
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. J. B. Wells and R. Vestergaard. Confluent equational reasoning for linking with first-class primitive modules. In *ESOP 2000 - European Symposium on Programming 2000*, number 1782 in Lecture Notes in Computer Science, pages 412–428. Springer, 2000.
23. 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.