UDC 004.438

ComponentJ: A Component-Based Programming Language with Dynamic Reconfiguration

João Costa Seco, Ricardo Silva and Margarida Piriquito

CITI – Centre for Informatics and Information Technology, and
Departamento de Informática, Universidade Nova de Lisboa, Portugal
joao.seco@di.fct.unl.pt, {ardoric, mpiriquito}@gmail.com

Abstract. This paper describes an evolution of the ComponentJ programming language, a component-based Java-like programming language where composition is the chosen structuring mechanism. ComponentJ constructs allow for the high-level specification of component structures, which are the basis for the definition of compound objects. In this paper we present a new language design for ComponentJ which is more flexible and also allows the dynamic reconfiguration of objects. The manipulation of components and composition operations at the programming language level allows for the compile time verification, by a type system, of safety structural properties of ComponentJ programs. This work is based on earlier fundamental results where the main concepts are presented and justified in the form of a core component calculus.

Keywords: Programming Languages, Dynamic Reconfiguration, Component Oriented Programming, Type Systems, Java.

1. Introduction

Component-Oriented Programming has been, for the past few years, an emergent programming paradigm. However, it has been supported mainly by low-level technological solutions which support the interoperability of independently developed objects. Traditionally, modularity in object-oriented languages has been based on the notion of class, and software reuse achieved by either implementation inheritance or object aggregation. On the one hand, inheritance is only useful in a limited set of scenarios and has been proved to become problematic in the context of large-scale software systems, as it may hinder evolution and interfere with issues such as dynamic loading [14]. On the other hand, object aggregation is usually implemented by ad-hoc idioms that manually build webs of objects.

This paper describes a component-based programming language, ComponentJ, which introduces, at the level of the Java framework, appropriate programming language abstractions and typing discipline for expressing the assembly, adaptation, and evolution of software components. ComponentJ was first introduced in [22] as the implementation of a basic model for typed

component programming. ComponentJ is presented in this paper with a new language design implementing the more general component model presented in [23,21] in the form of a core typed programming language whose first-class values are objects, components, and configurators. In this abstract programming model, *objects* are component instances (cf. class instances) which aggregate state and functionality in the standard object-oriented sense; *Components* are the entities that specify the structure and behavior of objects by means of a combination and adaptation of smaller components and user-defined building blocks. Each component is defined by a network of elements which is specified by a functional-like value, a configurator. *Configurator* values denote composition operations which aggregate or connect existing elements in an implementation independent way, and are uniformly used to produce components or modify the internal structure of objects. Thus, this variety of values and language constructs allows for both the expression of dynamic construction of new components (based on runtime decisions) and the unanticipated reconfiguration of component instances.

The ability to express component structures at a high-level of abstraction allows for the static verification of structural soundness of components and objects, by means of a type system. In particular, typing configurators with intensional type information, revealing certain aspects of their internal structure, permits type safe composition and reconfiguration actions to be performed on runtime values. Reconfiguration actions are, to some extent, a violation of the encapsulation principle ensured by the type system. In such circumstances, type safety is not entirely ensured at compile time, but is achieved by a combination of a static type checking and a light-weight dynamic compatibility check. ComponentJ's type system mimics that of the component calculus introduced in [23,21] and carries to the Java environment some of its key features, for instance, a structure-based type equivalence relation instead of the name-based type equivalence of Java. The base calculus also features bounded universal quantified types which are mapped into standard Java generic types.

The novelty of this approach when compared with other component models lays in the dynamic construction and runtime modification of the structure and behavior of objects in a statically typed Java-like setting. ComponentJ provides full computational power to build sophisticated and declaratively defined networks of objects while clearly maintaining the definition of architecture and computation separate. Existing approaches providing type safe assembly of components choose either to establish the construction of module structures at compile/link time [3,19] or limit the creation of extensible structures by defining fixed extension patterns [2]. Other component models and associated scripting and composition languages [16] have sophisticated runtime support systems and provide a wide range of composition mechanisms. However, component construction is based on programming conventions, reflection mechanisms and explicit control over the structure of objects using operational code. We focus on a more fundamental approach where the construction of new components and reconfiguration of objects and corresponding soundness properties are defined together in a unique programming lan-

guage and a single type system. This provides the language with a higher level of expressiveness and statically ensures the absence of runtime errors due to ill-formed component structures.

In the remainder of the paper we present ComponentJ's component model, its main ingredients and features. We next use a small toy example to illustrate the language constructs and the properties ensured by the type system. At the end of the paper, we relate this work with that of others and make some final remarks about the resulting implementation.

2. Component Model

The abstract model underlying the ComponentJ language design aims to provide a new structuring mechanism in the setting of an object-oriented programming language that safely expresses, at the programming language level, the programming idioms typical of the component-oriented programming style. This model is fully presented in [21] where it is instantiated in a typed component core calculus. In order to provide such a structuring mechanism it follows some fundamental design principles: making dependencies explicit, promoting the notion of dynamic construction of systems, support the modification of objects behavior at runtime, and guide the overall development by a typeful approach.

Dependencies between modules are implicit in most programming languages and are resolved at compile time for each particular module. This makes platforms based on dynamic loading of code prone to runtime errors due to missing modules. The first principle pursued in this model is to make explicit all dependencies between modules that would otherwise be inconspicuous in the code. This goal is achieved by representing modules as black-boxes with explicitly declared required and provided interfaces, where control over dependencies is placed at the programming language level. This approach resembles that of classic distributed systems defined in the style of Architecture Definition Languages [15,17] or the layout of electronic circuits where components are connected by wires. In both cases a system's architecture and the implementation of its components are treated separately. One of the important features of ComponentJ programming language is that it ensures the separation between the design of component's structure and that of computation by giving its runtime values a clear semantic interpretation.

Secondly, the abstract model aims at promoting the notion of dynamic construction of systems as a way of swiftly adapting and evolving an application to handle a large variety of situations. Therefore, the model states that the definition of functionality and that of structure should be placed at the same level, without compromising the separation of these concerns. Nevertheless, computational code should have full expressive power to guide the construction of new component structures.

Another goal of the abstract model is to support the unforeseen modification of the behavior of objects at runtime. The definition of language abstrac-

tions for composition operations allows for the definition and modification of components and objects to be uniformly expressed and controlled. This is achieved by defining small-grain composition operations and by mapping the structure of components to that of the runtime representation of objects. Such reconfiguration actions can be triggered in the course of computations and modify the network of elements and connections that define their target objects. Reconfiguration can be used in the case of an exception being raised by an object, or can be used in a planned way to implement software updates during runtime.

The last and most important guideline for developing the component model is typeful programming [9]. The representation of programming idioms, to build component structures, by means of high-level programming language abstractions allows for the application of type verification techniques to ensure good properties of composition and reconfiguration actions. Type systems for module languages usually ensure that module clients conform to their available interface information, that the implementation of modules satisfies the declared interface, and that they properly use other concrete modules. ComponentJ's type system also ensures that components are defined without inconspicuous references to external services and, therefore, that the resulting networks of connected objects are well formed. Furthermore, it ensures that the internal structure of dynamically built components is well defined and that reconfigured objects are sound. Intuitively, this corresponds to the absence of *method-not-found* and *null-dereferencing* runtime errors regarding references representing relations between objects in manually built webs of objects.

2.1. The Model Ingredients

The main ingredients of our model, which are also first-class values in ComponentJ, are *Objects*, *Components* and *Configurators*. The interactions between the values of the language are depicted in Figure 1.

Objects aggregate state and functionality in the standard object-oriented sense, and implement services (sets of methods) specified by standard interface types. In opposition to the notion of objects in class-based languages, which collapse a set of implemented services into one single provided interface, objects in ComponentJ provide a separate view for each service. These views are called ports and are identified by a name. Unlike other component-based programming languages [2,11], which refer to the object-like entities as components, objects in ComponentJ are component instances.

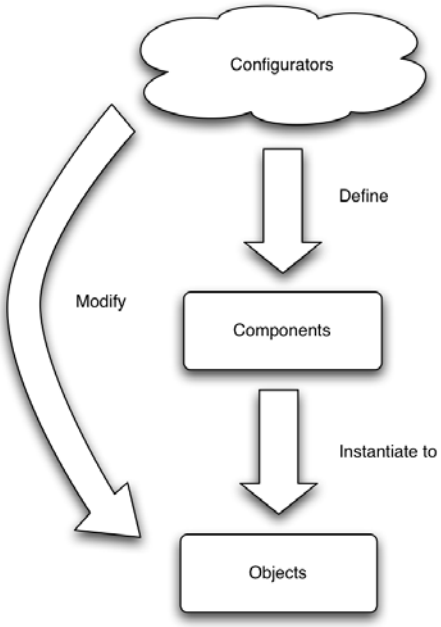

Fig. 1. Model Ingredients and Interactions.

Components are entities that specify the internal structure and behavior of objects. The implementation of services provided by component instances is defined by combination and adaptation of services provided by smaller components. At the component level, ports also play an important role as the connection unit between elements. A component declares a set of required ports, which denote abstract implementations of external services, and a set of provided ports, which it must implement. In this sense, the implementation defined in a component is parameterized in its required ports, which must be satisfied before the component is used to produce objects.

Finally, *configurators* are the basic building blocks of the language; they denote composition operations that describe the way in which components are aggregated and adapted to define other components or modify existing objects. The result of the application of a composition operation is called a component structure. Configurators define new scripting blocks implementing methods from scratch, introduce new elements into component structures, declare provided or required ports, or connect resources in a component structure. Configurators can also be combined to create other configurators that produce their joint effect. Unlike component composition, where only the required and provided ports are available to be connected in a *black-box* style, the elements introduced by two combined configurators bind without any visibility borders in a *white-box* style.

```
<componentDecl> ::= component C {<expression>}

<expression> ::= ...                          (Java expressions)
               | provides T p
               | requires T p
               | methods m { <declaration>* }
               | uses x = <expression>
               | plug <portname> into <portname>
               | <expression> ; <expression>
               | compose (<expression>)
               | new <expression> with [ p:= <expression>* ]

<portname> ::= x | x.p

<statement> ::= …                             (Java statements)
               | reconfig <expression> using <expression>
                     with [ p:= <expression>* ]
                     in <statement> else <statement>
```

Fig. 2. ComponentJ language syntax

3. Programming Language

In this section we illustrate the syntax and semantics of ComponentJ. We present the syntax of the language constructs and illustrate the language semantics using a simple step-by-step presentation of an example of a component implementing a counter.

3.1. Syntax

The syntax of ComponentJ, depicted in Figure 2, builds on an imperative fragment of Java, features top-level declarations for components, and high-level language constructs to express the manipulation of configurators, components and objects. In this syntax we assume a set of user-defined types whose identifiers are represented by the letter `T`, a set of component names represented by the letter `C` and we use `x`, `p`, `m` to denote locally defined identifiers.

The core of ComponentJ constructs is the set of expressions for basic composition operations on configurators (requires, provides, methods, uses, and plug) and a composition operation (c;c where expressions `c` and `c'` denote configurators). Each composition operation denotes a configurator, which is the kind of value expected as argument of expression compose, and in top-level component declarations, to define its internal structure. The next expression in Figure 2 is the instantiation expression (new) which has a sub-expression denoting a component and a list of port assignments (`p:=…`). Port

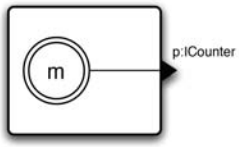

Fig. 3. Component Counter

assignments are the way to connect references of existing objects to newly created component instances. Notice that method blocks include a set of declarations of state variables and methods (cf. a Java class), which are written using the imperative fragment of the language.

We also introduce the statement (reconfig `o` using `c` …) that applies a reconfiguration action, denoted by expression `c` to an object denoted by expression `o`. The outcome of the reconfiguration statement depends on the result of a runtime procedure that checks whether the reconfiguration action is safe based on a small amount of runtime type information. The in branch of the statement is chosen if the reconfiguration is considered safe and the modifications have taken place. Any new provided ports added to the object are visible here. The else branch is chosen otherwise. Since reconfiguration actions may introduce new required ports, and they must be connected to some existing implementation, the reconfig statement also admits a with clause. Notice that whenever this list of port assignments is empty the with clause can be omitted.

We next illustrate the semantics of the language by means of an example.

3.2. An Example

We start by defining a component named `Counter`, implementing a service specified by a port interface `ICounter` at a port named `p`. In this example, we assume that there is an interface `ICounter` declared as having a method named `tick` receiving an integer argument and returning an integer as result. The functionality of component `Counter` is defined from scratch using only the basic building blocks of the language.

```
component Counter {
      provides ICounter p;
      methods m {
            int s = 0;
            int tick(int n) {
                  s = s + n;
                  return s;
            }
      }
      plug m into p;}
```

The top-level declaration above defines a component value statically associated with the identifier `Counter`. The structure of component `Counter` is depicted in Figure 3 where a black-triangle is used to denote a port at the component border and the solid line connecting it to the method block `m` denotes the explicit connection of the service implementation to port `p`. Component `Counter` is defined by a configurator value denoted by the sequence of composition operations its declaration encloses. The composition operations incrementally define its internal structure by declaring which elements it includes and how they are connected. A component definition establishes a visibility border around its internal components and building-blocks, thus limiting the communication with the outer context to well defined spots, the required and provided ports.

The first composition operation in this sequence, `provides ICounter p`, declares a provided port named `p`, which acts as a placeholder for an implementation of the service. The next composition operation defines an implementation of a service by means of the composition operation `methods m {...}`, a basic building block with the local name `m`, which includes state variables (`s`) and method implementations (`tick`). The implementation defined in method block `m` is connected to the declared port `p` by means of composition operation `plug m into p`. Method bodies are programmed using an imperative fragment of the Java programming language.

Component `Counter` is next used to produce an object using the instantiation operator `new`, and method `tick` is called at port `p` using standard dot notation.

```
o = new Counter;
o.p.tick(1);
```

Notice that calling method `tick` on port `p` implies actually calling method `tick` in method block `m`, which is explicitly connected to port `p` in the definition of component `Counter`.

Notice that component `Counter` above is a runtime value of the language; in this case the identifier `Counter` is permanently bound to that particular component. Next we define another component value by means of the composition and adaptation of existing components in a hierarchical way. The component value is here assigned to a variable named `ZeroCounter`. To do so, we use the composition operation `uses c = Counter` which introduces component `Counter` under the local name `c` as an element of the component structure being defined. An inner component is an element of a component structure where only it's required and provided ports are visible and can be referred by other composition operations (see Figure 4).

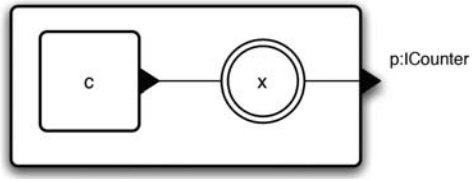

Fig. 4. Component ZeroCounter

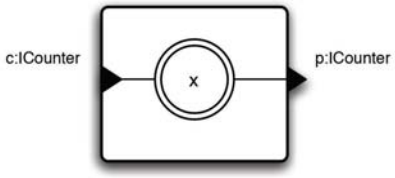

Fig. 5. Component ZeroFilter

```
ZeroCounter = compose (
      provides ICounter p;
      uses c = Counter;
      methods x {
            int tick (int y) {
                  if (y == 0) return c.p.tick(1);
                  else return 0;
            }
      };
      plug x into p );
```

Notice that the local name `c` for the provided port is bound to the occurrence of the identifier `c` inside method `tick`. Since composition operations are expressions evaluated separately and configurators are values of the language, we defined two separate name spaces which ensure the separation between computation and component structures. There are names which denote values of the language (objects, components, and configurators) which follow the standard name resolution strategy, and there are names visible in the context of compositions and which are bound by the explicit composition operation on configurators. An alternative to this kind of hierarchical composition, where `ZeroCounter` is defined containing component `Counter`, is to factor out the adaptation code in a separate component and composing it with a component `Counter` at a different abstraction level. We next define component `ZeroFilter`, whose internal structure is depicted in Figure 5, to intersect the calls to method `tick` made at a provided port `p` implementing a service `ICounter`

and redirecting only the calls made with 0 as argument to an external implementation available at a required port `c`.

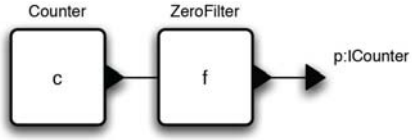

Fig. 6. Component Structure Using ZeroFilter

```
ZeroFilter = compose (
    provides ICounter p;
    requires ICounter c ;
    methods x {
          int tick (int y) {
                if (y == 0) return c.tick(1);
                else return 0;
          }
    };
    plug x into p );
```

The composition operation `requires ICounter c` declares a required port representing an external implementation of a service `ICounter`. This required port is introduced in the component structure under the local name `c`. Notice that before being actually used, required port `c` must be connected to some concrete implementation. This may be achieved by using component `ZeroFilter` in another component structure where a service `ICounter` is available and connected to port `c`. The expression denoting such a composition is depicted in Figure 6:

```
provides ICounter p;
uses c = Counter;
uses f = ZeroFilter;
plug c.p into f.c ;
plug f.p into p
```

This expression denotes a configurator that can then be used to produce a component.

A similar effect can be achieved by instantiating component `ZeroFilter` and providing a reference to an existing port (of an already instantiated component). Notice that a component can only be instantiated when all required ports are connected to concrete service implementations. In spite of this restriction, ComponentJ features a component instantiation mechanism that allows components with declared required ports to be instantiated as long as existing implementations are connected to existing ports of objects (instead of being composed with other components). This is a fundamental mechanism that allows sharing of object references between several instances of compo-

nents. The code instantiating and connecting the required services is the following

```
z = new ZeroFilter with [c := o.p];
```

Notice that we assume here the existence of object `o` with a port named `p` providing a service `ICounter`.

So far in the example, configurators are composition operations which define component structures used to define components. By choosing an adequate internal representation for objects at runtime, we are able to use the same composition operations on objects and in this way modify their internal structure. Consider configurator `addReset` below

```
addReset =
      provides IReset r;
      methods y {
            void reset() {
                  while (m.tick(1) > 0);
            }
      };
      plug y into r
```

Notice that the configurator `addReset` refers to a method block `m` implementing method `tick` which increments a counter with a given amount, and which is now used to decrement a counter until it reaches `0`.

An instance of component `Counter`, object `o`, can then be changed (and used) by means of the following expression

```
reconfig o using addReset in o.r.reset() else...
```

The **`reconfig`** statement applies the composition operation denoted by configurator `addReset` directly on the internal structure of object `o`. The application of the configurator `addReset` to an instance of component `Counter` has an effect which is approximate to the instantiation of a component defined by a compound configurator that adds the elements of component `Counter` and then the elements of `addReset`. Strictly from a structural point of view we can say that composition and instantiation commute through reconfiguration, the only difference being the modifications on the state of the objects. One of the strongest points of our approach is that reconfiguration and composition are defined uniformly using the same composition operations.

The typing discipline we present next ensures that reconfiguration is atomic, meaning that it is either fully applied, or it is not applied at all and the object's internal consistency is maintained. This demands for a runtime test to be performed before applying the reconfiguration operation to ensure that reconfiguration actions can proceed. The test is based on both static and dynamic type information which can be pre-processed to implement an efficient

test procedure, without the need for re-analyzing the source code. If the test succeeds, then it is guaranteed that both target object and reconfiguration action are conformant with the type checking performed statically and the reconfiguration can proceed. The internal structure of the target object is modified according to the given configurator and the object is available in the `in` branch of the statement possibly exhibiting new provided ports. If the test fails no reconfiguration is performed and an alternative action will be triggered in the `else` branch.

4. Typing

ComponentJ instantiates the type system defined for the core component calculus in [23,21]. The implementation keeps many characteristics of the base model, such as structure-based type equivalence up to a certain level, and extends the initial proposal with a new notion of subtyping between configurators. It also introduces some improvements related to the usability of the language, avoiding type annotations in some operations.

Besides a small set of primitive types, we define four different kinds of types: port interfaces, object interfaces, component interfaces and script interfaces.

Port interfaces type the services available in the ports of objects, defined by lists of method prototypes. In the case of the example presented in section 3, port interface `ICounter` is defined by

```
port interface ICounter {
      int tick (int n);
}
```

Object interfaces describe the type information of the services that objects implement. ComponentJ keeps the decoupling between typing and implementation introduced in the base language, for instance, the instances of component `Counter` have type

```
object interface OCounter {
      provides ICounter p;
}
```

which means that objects with type `OCounter` implement a service typed with `ICounter` located at a port named `p`. Based on this information we can already verify that the expression `o.p.tick(1)` is well typed: method `tick` is called on a service with type `ICounter` located at port `p` which is selected from object `o` with type `OCounter`.

Similarly, *component interfaces* also contain the type information about the provided services of a component. Additionally, they register the services required by components. For instance, component `Counter` has type

```
component interface TCounter {
      provides ICounter p;
}
```

this means that its instances have a type equivalent to OCounter above. On the other hand, the type associated with component `ZeroFilter` is

```
component interface TCounterFilter {
      requires ICounter c;
      provides ICounter p;
}
```

`TCounterFilter` declares that the components it types depend on an external implementation of a service with type `ICounter` on port `c` in order to implement a service of type `ICounter` on port `p`. Nevertheless, the instances of `ZeroFilter` are typed the same way as the instances of component `Counter`, with type `OCounter`.

Configurator values are typed more extensively; they are functional like values whose type reveals type information about the elements they manipulate. This allows for the verification of the flexible composition mechanism we presented for configurators (white-box) which is based on the nature of the elements they manipulate and their local names. Configurator values are typed with *script interface* types whose structure is split into two separate sections: introduced resources and needed resources

```
script interface S needs {
...
} introduces {
...
}
```

The first set of resources describes the elements needed by the composition operation, i.e. the preconditions for its application. The other set of resources describes the elements that remain visible for further composition after the configurator has been applied, i.e. the post-conditions of its application. A resource in a script interface indicates the presence (introduced or required) of an element in the component structure denoted by the configurator. It is expressed by a combination of a name (simple or compound), a role and a type. A simple name (e.g. `m`) may denote a method block, a required port, or a provided port in a component structure; a compound name (e.g. `c.p`) denotes a port of an inner component. Every entry in either the needed or introduced sections follows one of the forms

```
open T <portname>
available T <portname>
provided T <portname>
```

```
required T <portname>
```

The meaning of a resource depends on whether it is included in the **needs** section or the **introduces** section of the type. For instance, an *open resource* in the introduces section means that there is an unsatisfied dependency (e.g. the configurator **provides**), whereas if it is used in the context of needed resources, it means that in order for the configurator to perform its action some other configurator must introduce an open resource (e.g. the **plug** configurator). *Available resources* represent the implementation of a given service, which can be used in compositions with other configurators. Again, when used in the introduces set, it means that the current configurator holds the implementations for others to use, and when in the needs set, it means that some other configurator must introduce the resource so that it can be used. *Provided* and *required* resources denote ports that are either introduced or needed by the configurator. When two configurators are composed together, the resulting type is the composition of both types, where entries in the needed resources set are to be fulfilled by the other configurator introduced resources, and vice-versa.

Take the example of the configurator `provides ICounter p`, which is typed by the following script interface (`S`):

```
script interface S introduces {
      open ICounter p;
      provided ICounter p;
}
```

By adding a provided port to a component structure, this configurator introduces the need for a compatible implementation, denoted by the resource **open** `ICounter p`. The type also declares that this configurator adds a provided port, with type `ICounter` and name `p` which will appear in the interface of every component made from these configurators.

The configurator `plug m into p` has type `T` described as follows:

```
script interface T needs {
      available ICounter m;
      open ICounter p;
} introduces {
      available ICounter m;
}
```

The connection between two elements of a component structure has the precondition that the source of the connection must be available to be connected, that the target must be open for connection, and that the corresponding types must be compatible. Unlike in the base component calculus the syntax of the **plug** operation in ComponentJ is not type annotated. In this particular case, the configurator could be used to connect any two elements, named m and p,

with two compatible types. In ComponentJ we assign the configurator with the most general type possible.

When two configurators are composed, the resulting type is the composition of both types. The entries in the set of needed resources in the second configurator may be fulfilled by resources introduced by the first configurator; if not, the needed and introduced resources remain in the resulting type. By composing the two configurators

```
provides ICounter p; plug m into p
```

we obtain the type

```
script interface U needs {
      available ICounter m;
} introduces {
      available ICounter m;
      provided ICounter p;
}
```

This type describes the resulting configurator where a provided port is introduced, and its implementation is connected to a method block `m` not yet determined. The precondition denoted by the resource `open ICounter p`, introduced by the `provides` configurator, is satisfied in the compound configurator by the operation `plug m into p` (which has the corresponding postcondition). The presence of an available building block `m` which implements a service of type `ICounter` remains unresolved in the compound type.

The composition of configurator types is defined in such a way that the resulting type describes the combined effect of the two configurators. A condition imposed by the type system is that configurators used to build components must have an empty set of needed resources and must not introduce any open resource in the end. Thus, the resulting component has a sound structure. These properties over typing and semantics are expressed and proved correct for the component calculus in a subject reduction result and are expected to hold for ComponentJ [21].

The type system of the base calculus is adapted to ComponentJ by adding some type inference for plug operations and method blocks. Method blocks may reference ports, inner components or other method blocks from the component structure they are enclosed in. The type system needs to figure out what are the most general types for the unbound names that appear in method bodies, so that, at composition time, these names can be bound to the corresponding elements in a type-safe way. For plug operations, the type constraints necessary to connect the referred elements without instantiating them completely need to be maintained until the types of the concrete elements are determined.

A way to increase the flexibility of the language is by introducing a subtyping relation, as it provides a means of increasing functionality without having to change the original system [9]. This kind of relation is also established in

ComponentJ, allowing the comparison of both object and component types. The subtyping relation on object types is achieved by relating the different structures with each other. If, for instance, an object `a` provides a given port of type `t`, and another object `b` also provides a port of the same type `t`, and another of type `s`, then it can be said that the type of object `b` is a subtype of the type of object `a`, as it provides, at least, the same set of ports. When comparing component types, the subtyping relation is as intuitive as between objects, however, component types gather information not only about provided ports, but also regarding required ones. In this case, the set of required ports is compared contravariantly, and the set of provided ports is compared in a covariant way. Intuitively the principle of substitution between component values states that a component value can replace another if it provides at least the same services and if it requires at most the same services.

Typing reconfiguration actions requires more than static type information. Although the interface of a given object is taken into account at compile time, its internal structure is not known and cannot be used to type reconfiguration actions. In order to establish the safety of a reconfiguration action, static typing is complemented with a runtime verification based on type information gathered during the compilation process. The dynamic test compares type information stored within the object and configurator values and checks for their compatibility. The type system ensures that the changes in the interface of the target object are visible and its future uses are well typed. Since the type system detects and chooses whether to proceed with the reconfiguration action, this implies having two different branches in a reconfig construct. In one of the branches the object is visible with a modified interface, and in the other branch the object is left unmodified.

5. ComponentJ Compiler

Although ComponentJ is a full-fledged language its main expected usage is as a glue language for existing components which are then used in standard Java code. For the sake of simplicity, the ComponentJ compiler [25,1] translates ComponentJ code into Java code supported by a lightweight runtime support system. The compiler also generates special classes based on type information, stubs and skeletons, aimed at helping in the process of using ComponentJ components in Java and writing native components in Java.

In this section we will describe some of the implementation details of the runtime support system we developed to support the execution of ComponentJ programs, and in particular how the various entities in our model are translated to Java code.

5.1. Runtime Support System

The internal structure of ComponentJ's values is far richer than that of the objects in the Java virtual machine. The runtime support system designed to support ComponentJ programs is responsible for the manipulation of components and configurators as values and for the bookkeeping necessary in the reconfiguration of objects.

The translation of ComponentJ into Java code mimics the semantics of the base model [21] where objects contain four sets of internal elements, each denoting different kinds of elements: required ports, provided ports, instances of method blocks and internal component instances. In order to allow the unexpected modification of the internal structure of an object the compile-time names of component internal elements must be kept at runtime. This information is essential whenever an object is reconfigured. In the current unoptimized version of the compiler this information is also used whenever a port is accessed. Nevertheless, it is foreseeable that this penalty can be avoided and direct references from ports to implementations can be used.

Components are first class values that work as object factories. They are represented at runtime by Java objects containing a consistent sequence of composition operations, which are used to assemble and produce its instances. Components declared at top-level, statically bound to a name, are represented by singleton classes with the same name. The values associated with those components are similar to the dynamically created ones.

Finally, configurators are the first class entities representing the composition operations of the language. These operations range from the declaration of the building blocks of an object, an operation to connect some of these elements and a configurator composition operation. As first class entities in our model, they are also supported by Java objects at runtime. Configurators are applied to an object in order to change its internal structure, at instantiation and reconfiguration time, by direct manipulation of their sets of inner elements. By using objects to represent configurators we can create and compose new configurators and therefore support the translation of the sophisticated composition mechanism of ComponentJ.

5.2. Instantiation

A crucial part of the translation of ComponentJ to Java is to correctly code the instantiation process. The process of instantiating a component includes some delicate issues that must be taken into consideration. One of these issues is the order by which the internal components must be instantiated and connected. This is particularly relevant because the language allows the composition of mutually dependent components. We solve this issue by dividing the instantiation process into two phases: one phase where all instances are created unlinked and connected at the level of the present composition, and another where the connections are propagated throughout the hierarchy of

component instances. To support this two-phase assembly process we use placeholder objects for port implementations which are useful to define the connections without actually needing an already defined implementation. We call these placeholder objects *Port Providers,* and their job is to maintain information about the port it is connected to, regardless it is directly provided by a method block or indirectly provided by another port.

Once the first phase of the instantiation process is complete, all required internal objects are guarantied to exist, and all ports have a connection to their implementation. In the second phase of the instantiation process we use this information and follow all redirections so that all ports have a direct reference to the method block implementing their service. This phase is crucial to avoid following port redirections every time a port's service is accessed and ending up with uninitialized ports in the case of circular references.

5.3. Reconfiguration

In order to allow for the reconfiguration of objects we keep the structure of port providers of an object even after the instantiation phase. The information contained in these port providers is useful during the reconfiguration operations to allow references to be automatically updated whenever a port's implementation is changed. We perform these updates in a bottom-up fashion and we use a publish-subscribe policy to maintain a network of port providers that need to be notified when changes are made in a particular port.

5.4. Integration with Java Programs

The Java code obtained from compilation of a ComponentJ program is structured in a series of classes using the runtime support system at a low level of abstraction in the manipulation of the runtime representation of configurators, components, and objects. In order for ComponentJ to be well-integrated with the Java language we need configurators, components, and ComponentJ objects to be manipulated in a user-friendly and type safe way at a level of abstraction close to that of ComponentJ.

Integration goes both ways: to use ComponentJ components in Java programs and to use components programmed using the Java language inside ComponentJ programs. In order to do so our compiler generates two kinds of helper classes: *stubs*, which are used to help the programmer use ComponentJ components in Java, and *skeletons*, which are used to help the programmer in the process of programming in Java a component usable by ComponentJ.

Stub classes are based on the type information present in the ComponentJ code, and can be generated for each declared component interface. Stub classes also provide a wrapper around a ComponentJ component and allow the creation of instances with the help of a more natural and type safe inter-

face than the one available in the runtime support system. The resulting component instances are equipped with methods that directly access the services on each port as well as methods to access the corresponding port provider should it be needed as a requirement for the instantiation of other component instances. Stub classes also support the two-phase instantiation process described earlier in this section. Port assignments can be satisfied by calling special setter methods on the newly created object. The resulting object cannot be used before the second phase is complete.

Skeletons are abstract classes that take care of the internal representation of components and component instances and leave the implementation of the provided services to be specified by user defined subclasses. They are designed in such a way that almost all methods must be implemented by the user or else the Java compiler fails to typecheck the concrete class.

6. Related Work

It is commonly accepted that the central issues of component-based languages are modularity and code composition. These issues are notably isolated and studied in the field of mixin-modules [6,14,18]. The approaches using mixin-modules usually divide the languages in two layers: a first layer where modules are built and composed, and another layer, the base language, where the functionality of modules is defined. Composition is defined by binding of names and renaming operations instead of explicit connection of exported and imported names as in ComponentJ. One approach taken to implement Mixin modules in Java, SmartJavaMod [3], defines a meta-language to build and compose Java-modules. Unlike this approach, ComponentJ brings the composition operations to execution time and allows the module structure to be manipulated at runtime.

Other component based programming languages, such as ArchJava [2], use composition as a possible structuring mechanism at the programming language level. Both ArchJava and ComponentJ present mechanisms that allow the building of composite components, however, in ArchJava it is not possible, for instance, to export the behavior of internal components and to define new component structures at runtime. Unlike ComponentJ, whose components are used to instantiate objects, ArchJava's components hold state variables, implemented methods and communication ports. By doing so, dynamic construction of component structures is only allowed within pre-established connection patterns.

The Fractal component model [7,8] is a runtime support system that can be used both to define software architecture and functionality, as well as to perform dynamic reconfigurations of Fractal structures. These goals are achieved by means of a series of calls to a sophisticated support system or by using its underlying scripting language FScript. In Fractal, dynamic reconfiguration is achieved by introducing a control layer in the component definition, allowing external interfaces to introspect and reconfigure the components internal fea-

tures. The safety of these operations is not ensured, although some approaches to solving this problem have been suggested [16]. In ComponentJ, however, dynamic configuration is achieved with no need to perform introspection, and is ensured to be safe by a combination of static type checking and a lightweight dynamic verification.

One of ComponentJ's main features is the ability to perform dynamic reconfiguration of software systems, while ensuring that reconfiguration only occurs if no typing problems arise in the reconfigured system. Other approaches have been presented to reconfigure systems at runtime by manipulating modules at system level [26,5] and at the level of programming languages [24,12]. However, our approach allows the modeling of unanticipated reconfiguration of objects, which seems to be a new approach at the level of strongly typed programming language design. Reconfiguration is also a topic addressed in mixin-modules in a stateless context of modules [13] where composition operations can be interleaved with computation. Unlike this work we deal with an object-oriented and imperative setting.

SOFA (SOFtware Appliances) and DCUP (Dynamic Component UPdate) [20] were designed with the goal of dealing with common component-based programming and automated software downloading challenges, such as component updating at runtime, and silent software modification (minimum human interaction). Update operations in SOFA are very similar to the dynamic reconfiguration ones in ComponentJ, as templates (or configurator in ComponentJ) are applied to objects to change their functionality with no need to recompile the application. However, DCPU architecture introduces a new notion where components are split into two parts, permanent and replaceable parts, as well as into a functional and a control part. Updates only update the replaceable part of the component, replacing it with a newer version. The updating process is controlled by a component manager, which exists in the permanent part of the component, thus making the component itself responsible for how the updating process is performed. Unlike this approach, the reconfiguration mechanism in ComponentJ is designed to express unanticipated reconfiguration actions at the program level.

Other component-based models and architectures have also been studied, for application in specific domains, instead of general purpose ones. This is the case of, for instance, SCA (Service Component Architecture) [10], which provides a programming model (runtime system support) for systems based on a service oriented architecture. It advocates the principles of service composition and reuse: a system can be composed of new services specifically tailored for the intended application, as well as of components extracted from existing systems and/or applications. SCA provides support for a wide spectrum of programming languages and frameworks (e.g. BPEL, PHP, Java) and diverse communication mechanisms (e.g. Remote Procedure Call, Web services). The assembly model defines the system in terms of service components and composites. The former implement and use services; the latter describe the assembly of components from the point of view of its function. This includes connections between components/services and the references the system offers for its use. Again our approach lies on the design of fundamen-

tal type safe programming language mechanisms and not on the design of a runtime support system with rich tools to address composition in multiple scenarios.

7. Concluding Remarks

We present a Java-like programming language that captures programming abstractions related to the component oriented programming style. The design of ComponentJ inherits the four basic principles of an abstract component model presented in [21,23]: explicit dependencies between components, dynamic composition of new components, runtime reconfiguration of objects, and strong typing. ComponentJ was first introduced in [22] already featuring explicit dependencies between components and dynamic composition of new components and is improved by the work presented in this paper to also allow for the runtime reconfiguration of objects. These principles are captured in a programming model [21] and presented in a core component calculus which is closely followed in the design of ComponentJ. ComponentJ manipulates three kinds of values: objects which are the central computing entity, similar to objects in the object-oriented programming paradigm; components which are used to create objects, as classes in class-based languages; and configurators which are used to define components and change existing objects. The type system for ComponentJ ensures that components are well defined and that reconfiguration maintains the soundness of the reconfigured objects.

The prototype compiler presented in this paper translates ComponentJ programs to Java, which allows ComponentJ components to be used in Java programs, and also permits the definition of native components in Java.

8. References

1. The ComponentJ Site. http://ctp.di.fct.unl.pt/~jcs/componentj.
2. Jonathan Aldrich, Craig Chambers, and David Notkin. Architectural Reasoning in ArchJava. In European Conference on Object-Oriented Programming (ECOOP), pages 334-367. Springer, 2002
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.
4. Davide Ancona and Elena Zucca. A Calculus of Module Systems. Journal of Functional Programming, 12(2):91-132, 2002.
5. Gavin Bierman, Michael Hicks, Peter Sewell, and Gareth Stoyle. Formalizing Dynamic Software Updating. In On-line Proceedings of the Second International Workshop on Unanticipated Software Evolution (USE), April 2003.
6. Gilad Bracha. The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance. PhD thesis, University of Utah, 1992.

7. E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. An Open Component Model and its Support in Java. In Proceedings of the International Symposium on Component-based Software Engineering (CBSE'2004), Edinburgh, Scotland, 2004.
8. E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The Fractal Component Model and Its Support in Java. In Software Practice and Experience, special issue on Experiences with Auto-adaptive and Reconfigurable Systems, 2006.
9. Luca Cardelli. Typeful Programming. In E. J. Neuhold and M. Paul, editors, Formal Description of Programming Concepts, pages 431{507. Springer-Verlag, Berlin, 1991.
10. D. Chappel. Introducing SCA. Technical report, Chappell & Associates, 2007
11. Thierry Coupaye and Jean-Bernard Stefani. Fractal Component-based Software Engineering. In Object-Oriented Technology. ECOOP 2006 Workshop Reader, volume 4379/2007 of Lecture Notes in Computer Science, pages 117-129. Springer Berlin / Heidelberg, 2007.
12. Sophia Drossopoulou, Ferruccio Damiani, Mariangolia Dezani-Ciancaglini, and Paola Giannini. Fickle: Dynamic Object Re-Classification. In European Conference on Object-Oriented Programming (ECOOP). Springer, 2001.
13. Sonia Fagorzi and Elena Zucca. A Calculus for Reconfiguration. In Online proceedings of International Workshop on Developments in Computational Models (DCM at ICALP05), 2005.
14. Ira R. Forman, Michael H. Conner, Scott H. Danforth, and Larry K. Raper. Release-to-release binary compatibility in SOM. SIGPLAN Not., 30(10):426-438, 1995.
15. David Garlan. Software architecture: a roadmap. In ICSE '00: Proceedings of the Conference on The Future of Software Engineering, pages 91-101, New York, NY, USA, 2000. ACM.
16. Marc Léger, Thomas Ledoux, and Thierry Coupaye. Reliable dynamic reconfigurations in the fractal component model. In ARM '07: Proceedings of the 6th international workshop on Adaptive and reflective middleware, pages 1-6, New York, NY, USA, 2007. ACM.
17. J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying Distributed Software Architectures. In W. Schafer and P. Botella, editors, Proc. 5th European Software Engineering Conf. (ESEC 95), volume 989, pages 137-153, Sitges, Spain, 1995. Springer-Verlag, Berlin.
18. Henning Makholm and J. B. Wells. Type inference, principal typings, and letpolymorphism for first-class mixin modules. In Proceedings of the 10th International Conference on Functional Programming, pages 156{167. ACM Press.
19. S. McDirmid, M. Flatt, and W. Hsieh. Jiazzi: New-age components for old fashioned Java. 2001.
20. František Plášil, Dušan Bálek, and Radovan Janeček. SOFA/DCUP: Architecture for Component Trading and Dynamic Updating. pages 43-52. IEEE CS Press, 1998.

21. J. C. Seco. Languages and Types for Component-Based Programming. PhD thesis, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, 10 2006.
22. J. C. Seco and L. Caires. A Basic Model of Typed Components. In Elisa Bertino, editor, ECOOP 2000 - Object-Oriented Programming, 14th European Conference., number 1850 in Lecture Notes in Computer Science, pages 108-128. Springer-Verlag, 06 2000.
23. J. C. Seco and L. Caires. Types for Dynamic Reconfiguration. In Peter Sestoft, editor, Programming Languages and Systems, 15th European Symposium on Programming, ESOP 2006, number 3924 in Lecture Notes in Computer Science. Springer-Verlag, 03 2006.
24. Manuel Serrano. Wide Classes. In European Conference on Object-Oriented Programming (ECOOP). Springer, 1999.
25. Ricardo Silva. Compilador e Sistema de Suporte à Execução da Linguagem de Programação ComponentJ. Technical report, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, 2008.
26. Gareth Stoyle, Michael Hicks, Gavin Bierman, Peter Sewell, and Iulian Neamtiu. Mutatis Mutandis: Safe and Flexible Dynamic Software Updating. In Proceedings of the ACM Conference on Principles of Programming Languages (POPL), pages 183-194, January 2005.

João Costa Seco has received his B.Sc. in Computer Science by the Faculty of Science and Technology of the New University of Lisbon in 1993, and his M.Sc. and Ph.D. in Computer Science by the New University of Lisbon in 1997 and 2006, respectively. From 1996 to 2006 he has been a Teaching Assistant at the Computer Science Department of the Faculty of Science and Technology of the New University of Lisbon, and since 2006 he is an Assistant Professor at the same University. His teaching activities are in the area of Introductory Programming, Data structures, and Advanced Courses on Programming Languages and Compilers. His research activities are related to programming languages design and implementation. His PhD is focused on type systems for component-based languages. His current interests also include type systems for programming languages with primitive support for concurrency.

Ricardo Silva has received a B.Sc. in Computer Science by the Faculty of Science and Technology of the New University of Lisbon in 2008, and is currently doing his M.Sc. at the same university. His research activities are in the area of programming languages and models. Ricardo Silva is partially supported by the CITI/PLM/1000/2007 grant.

Margarida Piriquito has received her B.Sc. in Computer Science by the Faculty of Science and Technology of the New University of Lisbon in 2007. Currently she is enrolled in a M.Sc. in Computer Science Degree offered by the same University. Her research activities are in the area of design and implementation of programming languages.

Received: July 16, 2008; Accepted: November 19, 2008.