
Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

Dissertação de Mestrado

Mestrado em Engenharia Informática

# Type System for the ComponentJ Programming Language

Maria Margarida Lameira da Cunha
Piriquito (25954)

Lisboa

**Maio 2009**


Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

Dissertação de Mestrado

# Type System for the ComponentJ Programming Language

Maria Margarida Lameira da Cunha Piriquito (25954)

Orientador: Prof. Doutor João Costa Seco

*Dissertação apresentada na Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa para a obtenção do grau de Mestre em Engenharia Informática.*

Lisboa

**Maio 2009**

# Acknowledgements

By the end of this work it is easy to look back and find a certain number of people who, in one way or another, influenced what is being presented.

My first thank you goes to João Costa Seco. Thank you for your patience, for making me stress out whenever I needed to feel the pressure, and for always being available to discuss my progress.

I would never have managed to get through this work if it weren't for my family. Mom, dad, thank you for always listening, for letting me explain my thoughts even if you had no idea what I was talking about, and for that word of encouragement that you always know when to give. I would also like to thank my sister for letting me visit every once in a while, for letting me play with Rodrigo, and for always wanting to know how things were going. My gratitude also goes to my brother for inviting me over for dinner and taking my mind off work for a while.

Many other people were involved in my well being throughout this last year. In this group I would like to include Hugo and Anabela for letting me feel welcome in the office, and all the ones in IMG for letting me express my ideas in their white board and helping me overcome some problems.

My last thanks goes to Rui, who has always been there for me. Thank you for bearing with me even when I was flustered and annoyed. Thank you for making my days a little brighter, for the weekends you kept me company while I worked and for never complaining when I was too busy for you. It would have been way harder without your support.

# Abstract

With the constant evolution of software systems need arises for more structured implementations, where processes like software updates and changes in systems can be easily made, with no need to change what had previously been implemented. One possible solution to this problem is the use of component-based programming languages. This kind of programming languages tries to promote not only code reuse but also a black-box discipline where it is not needed how a service is implemented, but only its interface so that it can be used.

The ComponentJ programming language seeks to provide a simple way to perform component creation and composition, making this new programming paradigm somewhat easy to use. Because ComponentJ is meant to be an extension to the Java programming language it becomes possible to implement components using the whole expressiveness of this language. It is also possible, in ComponentJ, to dynamically change components and the object structure based on runtime decisions. This dynamic reconfiguration process allows, for instance, to perform changes/updates to a certain software system without having to stop its execution.

The goal for this project is to implement a type system for the ComponentJ programming language, based on the work presented in [32, 28]. Type verification is syntax driven, and uses structural equivalence of types. Advanced techniques such as subtyping and type inference are also included in order to make the language more flexible. Besides the static type checker, a dynamic checker is also included, allowing the type safe application of runtime changes to the system (dynamic reconfiguration of objects) before their application.

**Keywords:** Component orientation, type systems, dynamic reconfiguration, subtyping, type inference.

# Resumo

Com a constante evolução da complexidade dos sistemas informáticos torna-se necessário que as suas implementações sejam mais estruturadas, para que processos como actualizações ou extensões possam ser feitas facilmente e sem necessidade de alterar partes previamente feitas. Uma das possíveis soluções para este problema é a utilização de linguagens de programação orientadas por componentes. Estas linguagens promovem tanto a reutilização de código, como também uma disciplina de programação em caixa negra, não sendo necessário saber como é implementado determinado serviço, mas apenas que interface disponibiliza para o exterior.

A linguagem de programação ComponentJ procura disponibilizar um meio simples de criação e composição de componentes, facilitando assim a utilização deste novo paradigma de uma forma simples e segura. Tendo por base a linguagem de programação Java torna-se possível utilizar, na criação de componentes, toda a expressividade desta linguagem. É também possível, com o ComponentJ, alterar dinamicamente os componentes e a estrutura de objectos de um sistema, tendo em conta decisões de execução. Este processo de reconfiguração dinâmica permite, por exemplo, efectuar actualizações/alterações a um determinado sistema sem necessidade de parar a sua execução.

Com este trabalho pretende-se implementar um sistema de tipos para esta linguagem, baseado no trabalho apresentado em [32, 28]. A verificação de tipos é baseada na sintaxe, e utiliza equivalência estrutural de tipos. Técnicas avançadas como relações de subtipo e inferência de tipos são também incluídas para tornar a linguagem mais flexivel. Para além do verificador de tipos estático, existe também um mecanismo de verificação dinâmica para que qualquer alteração feita em tempo de execução (reconfiguração dinâmica) possa ser verificada.

**Palavras-chave:** Orientação por Componentes, sistema de tipos, reconfiguração dinâmica, subtipos, inferência de tipos.

# Contents

# List of Figures

# Chapter 1
# Introduction

In all well-established industries the use of pre-assembled components is widely common, all artifacts are usually built by assembling smaller third-party parts. For instance, building a car requires many smaller components: doors, windows, engine, breaks, etc.. So, if the concept of component and part assemblage is essential to almost every industry, why shouldn't it be used in software industry? why can't software programmers easily reuse third-party components to create larger software systems? The answer to these questions is that it should be possible. Programmers should be able to use previously created parts in their more sophisticated systems.

For some time now, software componentry industrial standards have been around, helping programmers in this task by providing programming conventions that allow the dynamic adhoc assembly of systems. The issue is that programming conventions are easily violated, making programing languages with primitive support for components the best approach to deal with that problem.

This work focuses ComponentJ, a programming language that supports composition and reconfiguration at the programming language level. An untyped implementation of a compiler prototype for ComponentJ already exists. This work is intended to provide an implementation for the lacking type system.

## 1.1 Background

### 1.1.1 From Objects to Components

The increasing pace of software industry, where solutions must be provided in a short span of time, demands programmers to produce reliable systems based on previously tested software pieces. The introduction of object-oriented programming languages

initiated a new approach to code reuse based on the notion of implementation inheritance, and provided modularity by means of classes.

One of the main issues of object-oriented programming languages is that achieving a consistent and coherent system architecture requires a great effort and discipline from the people involved in the programming process, as it tends to be very difficult for a programmer to avoid messing with the system's integrity for its own convenience. Software reuse by means of implementation inheritance tends to be insufficient, as it it only allows code extensions, but not the assemblage of heterogeneous components. Also, the connection between objects in object-oriented programming languages is implicit in the implementation code in most cases, making it hard to verify that systems have, indeed, the intended architecture.

The Component-oriented paradigm, on the other hand, states that system architecture and implementation are two different things, and should be treated differently. This separation is possible because components are seen as black boxes: their implementation is hidden from the outside context, and the only way to interact with them is through specific access points in the box (ports) that allows one to use specific exported services. In general, component parts are explicitly linked to each other using ports as the end-points of explicit connections, thereby defining a system architecture layer that can be read separately from the systems's code. The basic principle followed by software componentry is that one component implements a very specific service, and allows others to use it without having to know its implementation details.

Because of that, the system architecture depends only in the way components are linked to each other. Actually, components might be seen as building blocks, with different shapes and colors, but which fit together perfectly and that allow the creation of larger blocks. Components also come in different "shapes and colors", according to the services they implement, and can also be put together so that more elaborate services are provided. It's the way that components are put together that defines the architecture of a system.

In the end, the main difference between object-oriented and component-oriented paradigms is that components are all about encapsulating a functionality, while objects are all about encapsulating data, i.e. components are service oriented, while objects are identity oriented. This does not mean that both approaches are completely unrelated. Actually, components are usually implemented on top of object based systems as a

way of organizing and building them. As such, there is no components *versus* objects, but rather components *based* on objects.

Software componentry in main-stream programming frameworks is based on programming conventions and runtime systems that support the dynamic loading and binding of program modules (components). A more natural way to program software systems modularized this way and following the rules behind component-based programming is to use programming languages that allow doing so in the first place. Such programming languages should have primitive support for both component definition, and composition (building components by composing smaller components). As it is still an emergent paradigm, not many programming languages provide a useful way to deal with components, and when they do, it is mostly achieved by introducing additional programming conventions in object-oriented programming languages so that component modeling can be performed.

ComponentJ is an attempt to provide such programming language support. It introduces programming language constructs that allow component based operations to be expressed and verified. One of the main improvements introduced by ComponentJ is the explicit treatment of functional dependencies between modules. In most programming languages, dependencies between modules are implicit in the modules's code and only be resolved at linking time. In ComponentJ, however, all functional dependencies are explicitly expressed in the source code. By doing so it becomes possible to verify that every dependency is satisfied before running the program, thus avoiding runtime errors due to method-not-found errors, or missing components. ComponentJ also provides the ability to dynamically create and change the system's configuration (the structure by which objects are organized) depending on runtime decisions.

### 1.1.2 Type systems

Program verification tools try to ensure that software systems satisfy certain conditions and follow some specifications. Type systems are widely used tools that fall in that category, they are embedded in the compilers to ensure that certain kinds of execution errors do not occur during program execution.

Coming up with a good English definition of what a type system really is might be tricky. A definition can be found in [25] stating that *"A type system is a tractable*

*syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute"*. This definition also classifies types as being a kind of computable value resulting from the evaluation of a program using an abstract interpretation function. Types can be seen as some sort of organization into separate sets, where each set represents values that verify certain properties.

The use of type systems provides an organization of all possible data into specific "sections", according to their usage and behavior. As said in [15], *"A type may be viewed as a set of clothes (or a suit of armor) that protects an underlying untyped representation from arbitrary or unintended use. It provides a protective covering that hides the underlying representation and constrains the way objects may interact with other objects. In an untyped system untyped objects are naked in that the underlying representation is exposed for all to see. Violating the type system involves removing the protective set of clothing and operating directly on the naked representation."*. These sets can be as restrictive or as permissive as wanted, as long as all elements share the common properties.

According to [13], *"Types are essential for the ordered evolution of large software systems"*, as the use of types systems ensures the well-behavior of software systems. Most of modern programming languages incorporate some sort of type system, that can be either explicitly annotated (as it happens in Java), or implicitly annotated where most type information can be inferred, and there is no need for type annotations on the source code (such as ML).

A type system is used to keep track of the types given to every value in a program. As said before, each type has is own characteristics, and it is also the type system's job to make sure they are not violated, so that the program behavior remains predictable. In a more formal definition, type systems provide formal methods to help ensure that a system behaves according to some specification.

In the context of large software systems, where many programmers are involved for long periods of time, it becomes harder to ensure that the system will not fail. Bugs are often reported, maintenance is needed, new features are introduced. Until a stable version is obtained, the system is evolving. Type systems provide a means of ensuring that, at each stage of the development, certain errors will not occur at runtime, therefore increasing the reliability of evolving software systems [14].

The goal of this work is to extend the existing compiler implementation for the ComponentJ programming language [35] by introducing the implementation of a type

system that, besides the standard type system verifications, also checks component related operations (e.g. component creation and composition and dynamic reconfiguration).

## 1.2 Previous work

ComponentJ was first introduced in [30], in the form of a typed core programming language for expressing the assembly, adaption and evolution of software components. The language is formally expressed by means of a core typed programming language whose main first-class values are objects, components and configurators. *Objects* are component instances that have state and functionality, close to the primitive notion of objects in object-oriented programming languages. *Components* are entities that specify the structure and behavior of objects and finally, *configurators* are operations that aggregate and connect components in an implementation independent way, and which are used to produce new components or modify the internal structure of objects. With the use of these values and constructs, the expression of dynamic construction of new components and the unanticipated reconfiguration of component instances is allowed. A type system has also been studied for the core calculus, ensuring structural soundness of components and objects even after reconfiguration actions. The type system also includes an approach to deal with subtyping, recursive types and polymorphism.

The four primitive types used in ComponentJ's type system are: port interface, object interface, component interface and script interface. *Port interface* types are very similar to the Java notion of interface, and feature a set of method prototypes. *Object interface* type, used to type objects, represents all services provided in each port of the object. To gather all the needed information on components, when created a component is assigned a *component interface* type, where not only the provided ports are shown, but also all required ones. The *script interface* type, used to type configurators, is a more complex type, as it is separated in two sets of resources: one set denoting the elements that the configurator still needs to attaches to, and another describing the elements that remain visible for further composition. An extended explanation on these types will be given later on on this report.

A prototype of a ComponentJ compiler was introduced for a fragment of the language [31]. It was later on updated in [35] to include the complete untyped language, and implemented a runtime support system that allows the reconfiguration operations to be executed. The compiler translates ComponentJ constructs into pure Java code supported by a light-weight runtime support system. The resulting code is then compiled and used in a standard Java runtime environment.

## 1.3 Main Goals

The main goal for this work is to study and implement a type system for the ComponentJ programming language, based on the work in [28] and [35]. The type checker is split in two different parts: a static verification performed at compile-time, and a dynamic check performed at runtime. The static verification procedure ensures the good structuring of all objects created, while the dynamic checker is necessary to ensure that reconfiguration operations do not break the statically checked structure. Due to a strong notion of information hiding, where the internal structure of objects is hidden, this verification cannot be performed statically and is, therefore delayed to runtime and kept to a minimum set of verifications.

When transporting the type system from the component calculus to ComponentJ, some of the mandatory type annotations in the calculus syntax are omitted and inferred in ComponentJ. This makes the language more user friendly and easier to read. The type inference mechanism used in ComponentJ is based on the standard inference techniques introduced in [17] and adapted in this work to the new composition operations on configurators. Besides reporting the implementation of the typing procedure in ComponentJ compiler, this work also studies a new subtyping relation for configurators. This subtyping relation was not originally studied in the base calculus.

The Java programming language type system is name-based, i.e. every value used by the programmer will be assigned with a type, and different types are set apart by the name they are given. This causes some integration issues, as ComponentJ's type system is structure based, i.e., it only cares about the structuring elements of a specific type, and not how it is named. As so, moving from ComponentJ to Java is a careful process, ensuring that type relations are based on structure up to a point where maximum

compatibility level is reached and the connection to Java code is not lost.

Along with this work a set of ComponentJ examples are also provided to describe the type system and validate the implementation. The examples focus on specific operations of the programming language, and show how the type system react in those cases. This is used as a validation procedure, eventhough it is not an exhaustive process.

## 1.4 Structure of the dissertation

Chapter 2 presents some related works on component-based programming and how they integrate the concepts used in ComponentJ. The ComponentJ model and programming language is presented in chapter 3. A simple example is provided in this chapter to explain some features of the programming language, and how they can be used. Chapters 4, and 5 introduce the implementation details and principles of the ComponentJ's type system and the type inference mechanism. The proposal for a subtyping relation between configurators is discussed in chapter 6. On chapter 7 a set of examples can be found, used to test and therefore validate the type system implementation. The last chapter shows some concluding remarks about this work, as well as some future work.

## 1.5 Contributions

The work presented here benefited and contributed to the publishing of a paper in a national conference [33] and an extended version of the same in an international journal [34], defining a pragmatic version of the calculus and the implementation of the compiler and runtime system.

# Chapter 2
# Related Work

This chapter intends to present a survey of other research work related to this thesis. It presents several component-based models and programming languages. The comparison of other works with ComponentJ will focus three main characteristics of the languages:

- Representation of component-based concepts such as components and ports among others;
- Presence of a type system, and the properties it ensures;
- Presence of dynamic reconfiguration mechanisms at either architecture or implementation level (or both);

## 2.1 ArchJava

ArchJava [7, 8] is an extension to the Java Programming Language that allows the unification between the software architecture and its implementation using the same programming language, thereby simplifying the development process. The main purpose of ArchJava is to introduce a programming language to deal with components that guarantees communication integrity between architecture and implementation. A system has communication integrity if components only communicate directly with the components they are connected to in the architecture of component classes. This property ensures that the software system implementation respects the desired system architecture. The presence of a sound type system guarantees communication integrity between an architecture and its implementation, even in the presence of shared objects and run-time architecture configuration.

The three main ingredients of the ArchJava programming language are components, ports and connections. *Components* are obtained from the instantiation of *component classes*, and are seen as special objects that communicate with other components in a structured way. *Ports* represent logical communication channels between one component and the ones it is connected to, making it the only possible way for two components to communicate with each other. Ports declare three different sets of methods: *requires*, *provides* and *broadcasts*. The first set represents methods that are implemented by another component but which are available at the current port (this implies a connection between the component that implements the method and the one that uses it). Provided methods are implemented by the current component, and are made available to other components at the port that is being defined. *Broadcast* methods are very much like required ones except that they can be connected to more than one implementation, while required ones only allow the connection to a single implementation. The different port types are very similar both in ArchJava and in ComponentJ, however, in ComponentJ the notion of broadcast ports does not exist, as each port can only be connected to a single implementation.

This approach also supports hierarchical software architectures where components have got internal component structures (components connected to each other) to define their functionality. The outer components are called *composite components*, and the inner ones called *subcomponents*. Despite the possibility to create hierarchical composition, ArchJava does not present any mechanism to satisfy requirements of internal components via the component class itself, that is, any requirement of an internal component must be fulfilled with another internal component. Also, and unlike ComponentJ where inner components can connect with the outside via declared ports, ArchJava does not present ways for exporting the behavior of internal components to outside components.

Another interesting feature of ArchJava is the possibility to create dynamic architectures, i.e. architectures that change during the execution of the program, where new components can be dynamically instantiated and connected to each other. Unlike ComponentJ, whose components are used to instantiate objects, ArchJava's components are instantiated from component classes and hold state variables, implemented methods and communication ports, limiting the dynamic construction of component structures to pre-established connection patterns. As ComponentJ's components are treated as

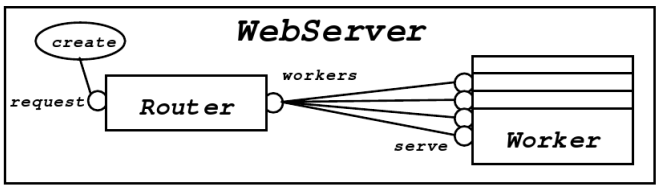


**Figure 2.1** Webserver Architecture in ArchJava

first-class values, computation over the structure of programs is allowed.

To better illustrate the model's capabilities consider the example depicted in Figure 2.1 which shows the architecture of a WebServer in ArchJava. The corresponding source code is presented in Figure 2.2. This example presents the definition of a WebServer, where *Router* is a subcomponent that accepts HTTP requests and passes them on to one element of a set of *Worker* components. When a request comes in, the *Router* requests a new worker connection on its *request* port. The WebServer then creates a new worker component and connects it to the *Router* at the port *workers*. The *Router* assigns requests to *Workers* through the established connection. This case illustrates both static architecture definitions (e.g. instantiation of component *Router*) and dynamic manipulation of the architecture (e.g. creation of a new component *Worker* for every request).

## 2.2 ComponentGlue

ComponentGlue [19] is the component-based language closer to ComponentJ, as its own implementation was based in the ComponentJ's model. It is used as a component composition language, that, despite ensuring most of ComponentJ's properties, introduces features to deal with distribution (by introducing mechanisms such as asynchronous communication and multicast), and ensures that the created executable compositions are consistent. As suggested by the component-oriented programming paradigm, ComponentGlue also follows the principles of software reuse, and is based on independent components that can be connected to each other.

One of the main differences between ComponentJ and ComponentGlue is that in ComponentJ, required ports are limited to one connection, i.e. one required port is connect to one and only one provided port (provided ports can be connected to as

```
public component class WebServer {
    private final Router r = new Router();
    connect r.request, create;
    connect pattern Router.workers,Worker.serve;
    public void run() { r.listen(); }
    private port create {
        provides r.workers requestWorker() {
            final Worker newWorker = new Worker();
            r.workers connection = connect(r.workers,newWorker.serve);
            return connection;
        }
    }
}
public component class Router {
    public port interface workers {
        requires void httpRequest(InputStream in,OutputStream out);
    }
    public port request {
        requires this.workers requestWorker();
    }
    public void listen() {
        ServerSocket server = new ServerSocket(80);
        while (true) {
            Socket sock = server.accept();
            this.workers conn = request.requestWorker();
            conn.httpRequest(sock.getInputStream(),sock.getOutputStream());
        }
    }
}
public component class Worker extends Thread {
    public port serve {
        provides void httpRequest(InputStream in,OutputStream out) {
            this.in = in; this.out = out; start();
        }
    }
    public void run() {
        File f = getRequestedFile(in);
        sendHeaders(out);
        copyFile(f, out);
    }// more method & data declarations...
}
```

**Figure 2.2** Implementation of a WebServer in ArchJava

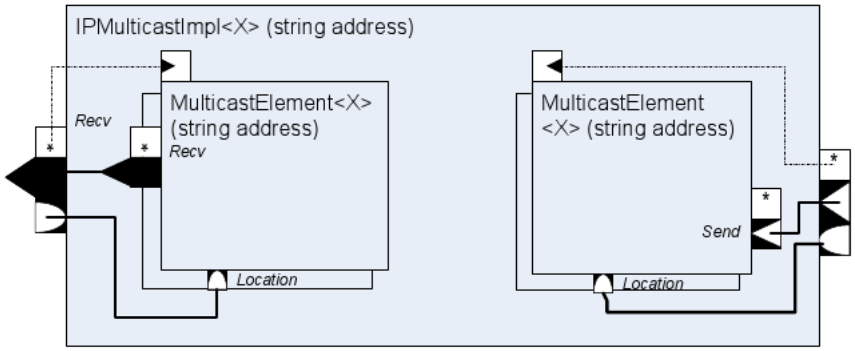


**Figure 2.3** *IP Multicast* generic component in ComponentGlue

many required ports as needed), while in ComponentGlue the limit restrictions to the number of connections allowed is determined by the kind of port is being used and its multiplicity. Another difference is that, as ComponentGlue is intended to be used in distributed environment, each component has an explicit given location.

In what concerns reconfiguration mechanisms, ComponentGlue does not allow dynamic reconfiguration of its structures and connections, but only the dynamic creation of instances and connections of compositions internal elements. ComponentJ, on the other hand, allows for the whole system to be dynamically reconfigured, by offering operations to alter the inner structure of components or objects, as well as their connections, based on runtime decisions.

Type compatibility in ComponentGlue is entirely based on structural equivalence, i.e. two different objects have the same type if all the operations performed by the first can also be performed by the second. Intuitively, the notion of subtype using structural equivalence is that a type A is subtype of type B if A includes all operations included in B, and possibly new ones. This is the base notion behind both ComponentGlue (in this case other details, such as links multiplicity, must be taken into account to define the relation) and ComponentJ (described later on this report) subtyping relation.

Since the motivation behind ComponentGlue is its usage on distributed environments, it becomes possible to implement communication mechanisms where semantics are expressed at the architecture level. The example in Figure 2.3 shows a generic *IP*

```
component interface TIPMSender<X> (string address) {
    port ? send {provides X}
}

component interface TIPMReciever<X> (string address) {
    port ? send {requires X}
}

component IPMulticastImpl<X> (string address) {
    port * send {
        provides X data_link;
        requires loc location_link;
    }
    port * recv {
        requires X data_link;
        requires loc location_link;
    }
    uses TIPMulticast<X>(string)senders[send]=IPMulticastElement<X>(address);
    uses TIPMulticast<X>(string)recvrs[recv]=IPMulticastElement<X>(address);
    plug send.data_link,send.location_link into senders.send,senders.location{
        send.datalink -> senders.send;
        send.location_link -> senders.location;
    }
    plug recv.data_link,recv.location_link into recvrs.recv,recvrs.location{
        recv.data_link -> recvrs.recv;
        recv.location_link -> recvrs.location;
    }
}
```

**Figure 2.4** Implementation of a *IP Multicast* generic component in ComponentGlue

*Multicast* architecture, whose source code is shown in figure 2.4. In this case the component *IPMulticastImpl* is implemented by means of two internal components defining sockets. Each of these components define two generic interfaces that allow sending and receiving asynchronous messages, and that are connected to external ports of component *IPMulticastImpl* (send and recv). The physical location of each component is defined by the location associated with each external port. Also, the component *IPMulticastImpl* is parameterized with a String that contains the multicast group address and the port of the receiver sockets.

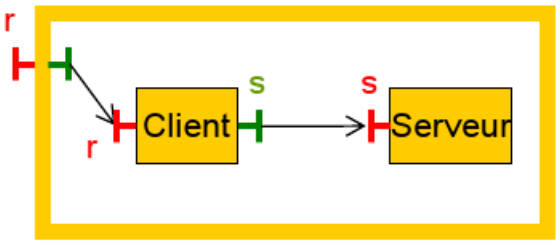


**Figure 2.5** Client-Server Architecture in Fractal

## 2.3 Fractal

The Fractal component model [11, 12] is a runtime support system that can be used both to define software architecture and computation, 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. This model is defined as being *"an extensible system of relations between selected concepts, where components can be endowed with different forms of control"*.

The main concepts behind the fractal component model are components and interfaces. *Components* are referred to as being encapsulated runtime entities with a distinct identity, while *interfaces* define access points to components. The concepts are very similar to the ones of object and port in ComponentJ. Depending on the level of control, the outside view of a component changes. At the lowest level of control components are seen as black boxes that allow no introspection, like objects in object-oriented programming languages, while at upper levels of control components can expose their internal structure, thus allowing introspection. At intermediate levels of control a component interface is provided, showing all its external interfaces. *Interfaces* support a finite set of operations, and can either be *server interfaces*, corresponding to access points that accept incoming operation invocations, or *client interfaces* which correspond to access points that support outgoing invocations.

Introspection provides a way to reconfigure the component's internal features. The safety of these operations is not ensured, although some approaches to solving this problem have been suggested [22]. In ComponentJ, however, no levels of control exist. All components and objects are seen and treated as blackboxes with specific provided

```
@Component(provides = @Interface(name="s",signature=Service.class))
public class ServeurImpl implements Service {
    public void print( String msg ) {
        System.out.println(msg);
    }
}

@Component(provides = @Interface(name="r",signature=Runnable.class))
public class ClientImpl implements Runnable {
    @Requires(name="s")
    private Service service;
    public void run() {
        service.print("Hello world!");
    }
}
```

**Figure 2.6** Implementation of a Client-Server Architecture in Fractal

ports that allow the interaction between the inside and the outside. Dynamic configuration is achieved with no need to perform introspection, and is ensured to be safe by a combination of static and dynamic type check.

Creating software using Fractal takes three steps: first, write the implementation code, second, add code annotation with Fractal metainformation, and last, write the linking code using Fractal Architecture Definition Language (ADL). The code annotations provide the information about classes that implement components and fields that represent required services, among others. To create a very simple client-server architecture such as the one in figure 2.5, the source code for both the server and the client components can be seen in figure 2.6. In this example there are two components *Serveur* and *Client* that are used as inner components to define another component. The *Serveur* component provides an interface *s*, while component *Client* provides an interface *r* and requires an interface named *s*, provided by *Serveur*. The linkage between provided port *s* in *Serveur* and required port *s* in *Client* is defined in the linking code (figure 2.7), building the system's architecture by binding client and server together. If instead of only one server two servers were needed, only the linking code would have to be changed. In this case, the resulting architecture would be the one in figure 2.8 and its linking implementation code the one in figure 2.9.

```
<definition name="HelloWorld">
    <interface name="r" role="server" signature="java.lang.Runnable" />
    <component name="client" definition="ClientImpl" />
    <component name="serveur" definition="ServeurImpl" />
    <binding client="this.r" server="client.r" />
    <binding client="client.s" server="serveur.s" />
</definition>
```

**Figure 2.7** Linking code using Fractal ADL

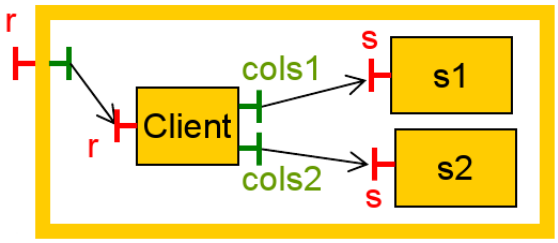


**Figure 2.8** Client-Server Architecture with two servers in Fractal

```
<definition name="HelloWorld">
    <interface name="r" role="server" signature="java.lang.Runnable" />
    <component name="client" definition="ClientImpl">
        <interface name="r" role="server" signature="java.lang.Runnable" />
        <interface name="cols" role="client"
                   cardinality="collection" signature="Service" />
        <content desc="ClientImpl" />
    </component>
    <component name="s1" definition="ServeurImpl"> ... </component>
    <component name="s2" definition="ServeurImpl"> ... </component>
    <binding client="this.r" server="client.r" />
    <binding client="client.cols1" server="s1.s" />
    <binding client="client.cols2" server="s2.s" />
</definition>
```

**Figure 2.9** Linking code using Fractal ADL

## 2.4 JavaBeans

Another existing component model for the Java Programming Language is JavaBeans [4]. In this model, beans are described as being *"reusable software components that can be manipulated visually in a builder tool"*. The idea behind beans is that, for software reuse purposes, it is easier to distribute a single object that encapsulates several others inside instead of multiple single objects.

The common, but not the only one, usage for beans is to build graphic user interfaces (GUI), as each and every different part of a GUI can be seen as a component, and the whole GUI is the connection between all other components. This usage of components appears in the major GUI toolkits, such as AWT [1], Swing [3] and SWT [5]. All these toolkits provide both a set of pre-created components that can be used and adapted to the programmers needs, and layout managers to combine components with each other. With the success it has reached, many large companies ended up adopting Enterprise JavaBeans [2], an extension to the JavaBeans component model used to handle the need of transactional business applications, as a technological solution.

JavaBeans is a model that works on top of Java, where beans are Java classes that follow certain conventions about method naming, construction and behavior, being that it's main concepts are: introspection (discover a bean's features, i.e. its properties, methods and events), properties (appearance and behavior characteristics of a bean), events (the way beans communicate with each other) and methods (java methods that implement beans). ComponentJ, on the other hand, is a programming language by itself, making the use of components more intuitive and simpler.

## 2.5 SmartJavaMod

A module is intended to be a unit structure that allows code reuse in a type safe way. The concept of mixin-modules was first introduced in [18] as new constructs for module languages, intended specifically for the Standard ML language. They add features to the language like mutually recursive modules, and virtual module components that can later be redefined by means of an override operator.

Module systems are intended to present features like interfaces, so that there is no

need to look any further than the specification, separate type-checking, where modules are type-checked in isolation and independently from other module, and module expressions that allows modules to be connected and combined with each other.

As object-oriented programming began to increase in popularity, attempts to define a module systems on top of them became a subject of study. One approach to design a typed module system on top of Java language is JavaMod [10]. The most important break through it presents is the design of a typed module system integrated with the Java type system. Although, for this type system to work properly, the programmer has to explicitly annotate with types the classes used as parameters in modules, making the type system too restrictive. To overcome this JavaMod limitation, Smart-JavaMod [9] was introduced, featuring a richer set of module operators, as well as an even more expressive type system that performs type inference so that module types can be omitted.

As said before, modules present a way to allow code reuse, and, with its usage in object-oriented languages, encapsulation is also achieved, as specification is separated from implementation. This notion is very close to the one of components used in ComponentJ, making components a natural evolution from modules. ComponentJ takes on, besides the advantages of using components, some of the advantages of module usage, mainly the typing techniques, and the ability to deal with mutually dependent components (mutually recursive modules).

## 2.6 SOFA and DCUP

SOFA (SOFtware Appliances) and DCUP (Dynamic Component UPdate) [26] 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).

In SOFA, an application is viewed as a hierarchy of software components, where software components are introduced as instances of component templates, or just template. These concepts relate to the ones of object and component used in ComponentJ. A *template* is a framework containing definitions of implementation objects and nested

```
template P:Bank (Property:num_of_tellers){
  provides:
    TellerInterface teller[num_of_tellers];
    SupervisorInterface supervisor;
}

template P:Supervisor {
  provides:
    SupervisorInterface supervisorAccess;
  requires:
    DataStoreInterface dataStoreAccess;
}

architecture P:Bank version v1 {
  auto inst P:Supervisor version v4 s;
  inst P:DataStore version v3 ds;
  bind s.dataStoreAccess ds.dataStoreAcess;
  bind supervisor s.supervisorAccess;
  for n=1 to num_of_tellers {
    bind teller[n] local;
  }
}
```

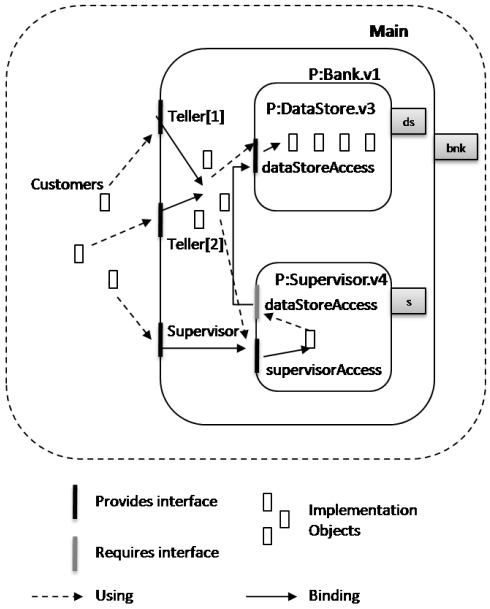


**Figure 2.10** Example of a SOFA template

components, and it's determined by an interface that specifies its provided and required services, and by bindings of implementation objects and nested components. An example of a SOFA template can be found in figure 2.10, where template *P:Bank* is defined by means of two other templates: *P:Supervisor* and *P:DataStore*. The way templates connect to each other is defined in the *architecture* block. Notice that each template used in the architecture is associated with a version. This allows for the update manager to keep track of changes.

Update operations in SOFA are very similar to the dynamic reconfiguration ones in ComponentJ, as templates (or configurators 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.

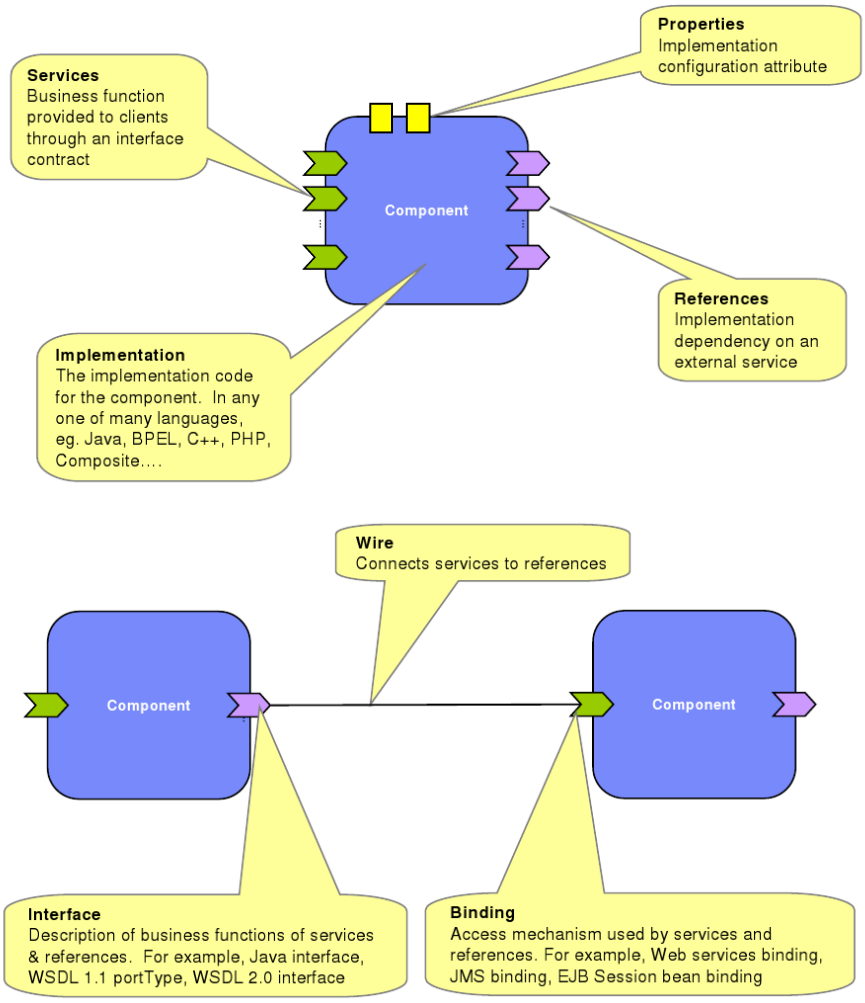


**Figure 2.11** Visual Representation of SCA Concepts

## 2.7 SCA

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 SCA (Service Component Architecture) [16], 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.

```
@Service(AccountService.class)
public class AccountServiceImpl implements AccountService {
    private String currency = "USD";
    private AccountDataService accountDataService;
    private StockQuoteService stockQuoteService;

    public AccountServiceImpl(
            @Property("currency") String currency,
            @Reference("accountDataService") AccountDataService dataService,
            @Reference("stockQuoteService") StockQuoteService stockService) {
        this.currency = currency;
        this.accountDataService = dataService;
        this.stockQuoteService = stockService;
    } // end constructor

    public AccountReport getAccountReport(int customerID)
            throws AccountDataUnavailableException {
        AccountReport accountReport =
            accountDataService.getAccountReport(customerID);
        List<Stock> stocks = accountReport.getStocks();
        List<StockValues> stockValues =
            stockQuoteService.getValues(stocks,currency);
        accountReport.setStockValues(values);
        return accountReport;
    }
} // end class
```

**Figure 2.12** Annotated Source code using SCA

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. Other concepts in SCA are the ones of *Wires* that connect services to references, *Interfaces* that provide a description of both services and references, and, at last, *Binding*, which introduces an access mechanism used by services and references. A visual representation of these concepts can be seen in figure 2.11.

The use of SCA within programming languages such as Java is very similar to Fractal [22], and is performed by introducing annotations into the source code for SCA elements. An example of annotated source code can be found in figure 2.12, representing part of an accounting system. The annotation *@Service* means that a new service interface is being defined,where each method implementation represents one of the methods provided by the service.

# Chapter 3
# ComponentJ

***component:*** *one of the separate parts of a machine or a system,*
*that is necessary to make the machine or system work.*

ComponentJ is a component-oriented programming language, introduced in [30, 29], based on a core component calculus in [28]. As said before, when dealing with components, basic language constructs are introduced so that component creation and connection can be easily achieved. ComponentJ is meant to be a glue language, built on top of the Java programming language, where not only creation and connection operations are allowed, but also dynamic reconfiguration. In this chapter the abstract model underlying the ComponentJ language is presented, as well as the ComponentJ programming language itself. The programming language usage is explained by means of a simple example.

## 3.1 ComponentJ Model

The abstract model underlying the ComponentJ language is fully presented in [28], where it is instantiated in a typed component core calculus. Its 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. To provide such a structuring mechanism, some design principles must be followed: 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.

### 3.1.1 Design Principles

Dependencies between modules are implicit in most programming languages and are only 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. To solve this problem, the first principle pursued in our model is to make explicit all dependencies between modules, that would otherwise be inconspicuous in the code. This goal is achieved with a black-box representation of modules, where all required and provided interface are explicitly declared, and also 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 [20, 23] or the layout of electronic circuits where components are connected by wires. In both cases architecture and implementation are seen as two different things, and are therefore treated separately.

The second design principle is the promotion of the notion of dynamic construction of systems which seems to be unexplored at the level of programming language design. The assembly of systems at runtime, depending on runtime environment information, is a way of swiftly adapting and evolving an application to handle a large variety of situations. This means that the definition of both functionality and structure should also be placed at runtime level, without compromising the separation of concerns.

Another goal of our model is to support the modification of the behavior of objects at runtime. The definition of language abstractions for composition operations allows 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 and the network of elements and connections that define them. Such reconfigurations can be triggered in the course of computations, or 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 [13]. 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. Typing in module languages usually ensures

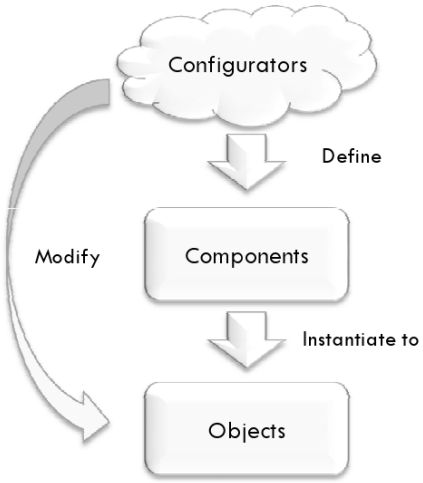


**Figure 3.1** Model ingredients and interactions

that module clients conform to their available interface information and also that the implementation of modules satisfies the declared interface and properly uses other concrete modules. ComponentJ's type system ensures that components are defined without inconspicuous references to external services and, therefore, that the resulting networks of connected objects are well-formed. It also 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.

### 3.1.2 The Main Ingredients

According to the concepts behind component-based software, the main ingredients of this model are objects, components and configurators. These ingredients are all treated as first-class values and interact as it is shown in Figure 3.1. *Objects* are component instantiations (much like in object-orientation objects are instantiations of classes) that aggregate state and functionality in a common object-oriented way. Each object can implement several interfaces, providing a different view, or port, for each service. *Components* are entities that specify the internal structure and behavior of objects. A

component declares a set of provided and required ports (ports are seen as connection units between elements), where the first set denotes all the services the component must implement, and the second one an abstract implementation of external services that can be used. *Configurators* are used to describe the way components are aggregated and adapted. They can be used to describe new scripting blocks, introduce new elements into component structures, declare provided and required ports, or connect resources already introduced. Configurators can also be combined with each other, producing a configurator with their joint effect.

The next section illustrates the syntax and semantics of ComponentJ. The syntax of the language constructs is presented and illustrated by means a simple step-by-step example.

## 3.2 ComponentJ Programming Language

ComponentJ is a component-based programming language based on the component model described before. A prototype compiler for ComponentJ has been implemented [35], making it possible to use ComponentJ as a standard programming language. The compiler translates ComponentJ source code into Java code that can be compiled with a standard Java compiler and used as any other Java program.

One main feature available in ComponentJ, and not commonly seen in other component based programming languages, is the ability to perform dynamic reconfiguration in a computation dependent and type-safe way, i.e. changing object's internal structure, but also modifying the connections between those elements, based on runtime information. In ComponentJ, reconfiguration is achieved by applying a configurator script to an object. Type checking this operations is not a trivial matter due to the strong notion of information hiding present in the language. The approach taken to solve this problem is discussed in the next chapter.

### 3.2.1 The syntax

The syntax of ComponentJ (figure 3.2) builds on an imperative fragment of Java, featuring top-level declarations for components, user-defined types and high-level language

```
<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>
```

**Figure 3.2** ComponentJ language constructs.

constructs to express the manipulation of configurators, components and objects. In this syntax we consider a set of user-defined type identifiers *T*, component names *C* and we use *x,p,m* for user-defined identifiers.

The core of ComponentJ constructs is the set of expressions for basic composition operations (*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 the expression *compose*, and in top-level component declarations, to define its internal structure. The next expression in figure 3.2 is the instantiation expression (*new*) which has a sub-expression denoting a component and a list of port assignments. Port assignments are the way to provide 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 Java fragment of the language.

We also introduce a statement that applies a reconfiguration action (a configurator) to an object which has two branches. Depending on the outcome of the runtime test prior to the reconfiguration one of the branches is selected. The *in* branch is chosen if

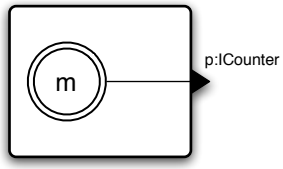


**Figure 3.3** Component *Counter*

the reconfiguration has taken place. Any new provided ports in the object are visible here. The *else* branch is chosen otherwise. Since reconfiguration actions may introduce new required ports, the *with* clause of the instantiation expression is also present. 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 a simple example implementing a counter.

### 3.2.2 An Example in ComponentJ

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 3.3

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. We next define another component value by means of the composition and adaptation of existing components in a hierarchical way. The component 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 its required and provided ports are visible and can be referred by other composition operations (see figure 3.4).

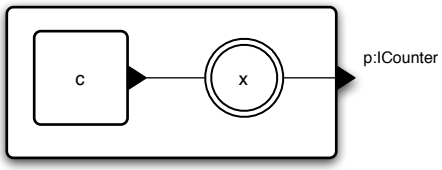


**Figure 3.4** Component *ZeroCounter*

```
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 3.5, 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*.

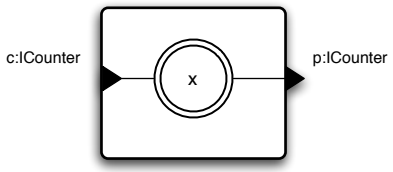


**Figure 3.5** Component *ZeroFilter*

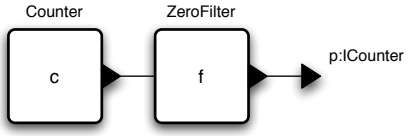


**Figure 3.6** Component structure using *ZeroFilter*

```
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 as follows, and can be visualized as shown in figure 3.6:

```
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 (already instantiated). 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 provided 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 components. 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 value *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 preprocessed 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.

# Chapter 4
# A Type System for ComponentJ

***type:*** *one of a group or class of people or things*
*with similar features or qualities.*

The existent prototype for the ComponentJ compiler [35] does not feature a fully functional type system. Nevertheless extra type annotations must be introduced by the programmer in the source code, so that the compiler can generate the correctly typed Java code. Indeed, no type verifications are performed, so the responsibility of writing well typed programs lies on the programmer alone.

This work aims at adapting and providing an implementation of the type system of [28] and extending the existent compiler prototype to guarantee that no execution errors will occur due to typing mistakes. We also allow most of type annotations to be omitted and the corresponding typing performed by inference, thus increasing the usability of the language.

## 4.1 Untyped ComponentJ

The prototype compiler which is the base for this work requires that type annotations are introduced in the source code in order to generate well-typed Java code. When calling a method on a port of an object one needs to introduce a type cast to the intended type of the port so that this cast is copied to the target Java code. The correct type coercion from a general port type to a concrete interface type is then achieved. Also, when defining a method block, one needs to explicitly declare all external references, by means of an *import* clause. These now unnecessary type annotations are not used for verification purposes but only for code generation.

To show the differences between the untyped and typed versions of ComponentJ, we next present the source code of the component structure depicted in Figure 4.1. The example features two different components, *C* and *D*, both providing a port named *p*.

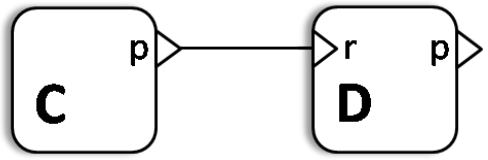


**Figure 4.1** Architecture of two ComponentJ components connected together

Component *D* also requires a service implementation at a port named *r*. To begin with, we define port interfaces types *I* and *J* used in the example as follows:

```
port interface I {
    void set();
}

port interface J {
    int get();
}
```

We next define the implementation of both components. Since no unknown resources are referred to by the method, both typed and untyped versions of the compiler component *C* are defined by the code that follows. Component *C* provides a port named *p*, with the implementation provided by a method block named *m*.

```
component C {
    provides J p;
    methods m {
        int a = 0;
        int get() {
            return a;
        }
    };
    plug m into p;
}
```

Component *D*, however, requires an external service implementation, thus making the untyped and typed implementations different. The implementation of component *D* is the following:

```
component D {
    provides I p;
    requires J r;
    methods m {
        import r : port J;
        int a;
        void set() {
            a = ((J)r).get();
        }
    };
    plug m into p;
}
```

*Untyped implementation*

```
component D {
    provides I p;
    requires J r;
    methods m {
        int a;
        void set() {
            a = r.get();
        }
    };
    plug m into p;
}
```

*Typed implementation*

In the untyped version, the type information about the required port *r* is needed to produce well-typed code for the statements inside method block *m*. So, the programmer must introduce a type cast for the expression representing a method call to indicate the type of the port being used, as well as an *import* clause stating that an external port named *r* of type *J* is being used. The need to introduce such information is useless and cumbersome for the programmer. Besides having no verifications associated, it makes the source code more complex and harder to read. In the presence of a type system implementation, both situations (cast and explicit name declaration through *import*) are resolved without extra annotations in the source code. The types associated to names declared outside method blocks can be deduced and inferred from the rest of the source code.

A very similar process is needed when calling a method of an instantiated object at a specific port. In this case, a type cast for the type of the port is also required in the untyped version. As before, that type annotations can be omitted in the present implementation. Consider the code fragment instantiating two objects:

```
o = new C;
oo = new D with [r := o.p];
```

In order to call method *set* on port *p* of object *oo* it is necessary for the programmer to provide type information about the accessed port. In the untyped version, calling

```
port interface type_name {
    method_prototype_1
    ...
    method_prototype_n
}
```

**Figure 4.2** Syntax of a port interface type

method *set* must be written:

```
((I)oo.p).set();
```

while in the typed version, as the type of the port can be deduced, that same expression is written:

```
oo.p.set();
```

As expected, the removal of unnecessary type annotations from the source code is the least of the improvements accomplished by the implementation of the type system. The main improvement resides in the type verification process of source code. These verifications ensure the structural correctness of any component and object in well-typed programs. We illustrate the typing process by means of an example and explain the implementation details in the next couple sections.

## 4.2 Typed ComponentJ

The type system of the ComponentJ programming language, defined in [32, 28] and implemented in the course of this dissertation, includes techniques like type inference and type abstraction. The type system uses four different kinds of types, representing the four different kinds of values of a ComponentJ program: port interface, object interface, component interface and script interface. Type equivalence is based on structure rather than names, i.e. two types are equivalent if their internal structure is the same.

*Port interface* types, whose syntax is depicted in Figure 4.2, are defined by a set of method prototypes. *Object interface* types are assigned to every object and contain

```
object interface type_name {
    provides port_type port_name_1;
    ...
    provides port_type port_name_n;
}
```

**Figure 4.3** Syntax of an object interface type

```
component interface type_name {
    provides port_type port_name_1;
    ...
    provides port_type port_name_n;
    requires port_type port_name_n+1;
    ...
    requires port_type port_name_m;
}
```

**Figure 4.4** Syntax of a component interface type

information about all the services they implement. This information is retrieved directly from the type of the component used to instantiate the objects. The syntax of this kind of type is shown in Figure 4.3. Similarly to object interface types, *component interface* types, as shown in Figure 4.4, also keep information about services provided by components, although, they also keep track of the required external services needed to implement the required services. The most elaborate type in ComponentJ's type system is the *script interface* type, which is assigned to all configurators. These have a more complex structure. They are split into two separate parts: introduced resources and needed resources, as can be seen in Figure 4.5. Resources are bits of information that represent the presence of certain conditions in partially built component structures. Every resource in either the needed or introduced sets has one of the forms represented in figure 4.6.

The semantics of a resource depends on whether it is included on the needs set or the introduces set of a script interface. For instance, an *open resource* used in the

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

**Figure 4.5** Syntax of a script interface type

```
open type name
available type name
provided type name
required type name
```

**Figure 4.6** Script interface resources

introduces set means that there is an unsatisfied dependency of a certain type and a given name (e.g. when the configurator *provides* is used to introduce a new provided port), while 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. when the plug configurator is used). *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 implementation of a set of methods that can be used by others, and when in the needs set, it means that it needs an actual implementation that some other configurator must introduce. *Provided* and *required* resources denote ports that are either introduced or needed by the configurator. When a composition operation is performed between configurators, the resulting type is the composition of both types, where entries in the needed resources set of one type are to be fulfilled by the other configurator's introduced resources.

Components are defined using configurators, and it needs to be ensured that after the composition of all of its configurators, the resulting script interface type will have no entries in the needs section, and no open resources in the introduced resources. This

verification ensures the well structuring of the component.

ComponentJ features dynamic reconfiguration operations, and performing static type checking for these reconfigurations is not possible, as there is no way to know the exact structure of objects at a given time (they can change during the execution of the program, based on runtime decisions). To overcome this problem, but to keep allowing dynamic reconfiguration to be performed in a type-safe way, a dynamic type test has been introduced in the language, ensuring that changes will only take place if no structuring problems arise from them.

Also, for some verifications to take place, some type information is omitted, making way for a type inference mechanism to exist. When performing a composition operation between two components, some information might not be directly available, mainly in what concerns method blocks and which port interfaces they implement, and external resources whose methods are used in the method implementation.

Type verification also takes into account possible subtyping relations between values. In ComponentJ, type equivalence is based on structure and not on name, so the equivalence and subtyping relations are achieved by relating the different type 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 one of object $a$, as it provides the same set of ports, and even more. If two objects are related by a subtyping relation, then, intuitively, the execution should not fail if the object of the subtype is used instead of the object typed by the supertype.

The subtyping relation between components is as intuitive as the relation between object types. Component types gather information not only about provided ports, but also regarding required ones. In this case, a component type can be said to be subtype of another when it provides, at least, the same ports, and requires, at most, the same ports as its supertype. If the required ports verification wasn't performed, when using a sub component, required dependencies could not be fulfilled, leading to a runtime error. For instance, take component interface types $A$ and $B$:

```
component interface A {
    provides I p;
    requires J r;
}
```

```
component interface B {
    provides I p;
    provides J q;
}
```

In this case we can state that there is a subtyping relation between type $A$ and type $B$ where $B$ is subtype of $A$. This relation is possible because component $B$ still provides a port named $p$ with the same type $I$, just as type $A$ does, while not requiring any external port. In a component structure, replacing a component typed with $A$ by a component typed with $B$ could be done, since any connection to the provided port would not be lost. Loosing the connection to the required port (since it ceases to exist) is not a problem, since that connection is no longer necessary.

In what concerns configurator types, and since their structure can easily become very complex due to the composition operation between configurators, defining a subtyping relations can be tricky and counter-intuitive. As part of the goals of this thesis, and since this relation was not established in the base calculus [28], a discussion about this topic is presented in chapter 6.

## 4.3 A typed example in ComponentJ

The best way to understand the steps taken by the type system is to include them incrementally in an step by step example. This section is based on the example used in Section 3.2.2 and includes the type verifications performed throughout the execution.

The first step in the example is to create the port interface *ICounter* that will provide the necessary methods to interact with a counter:

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

This declaration adds to the typing environment a binding between the name *ICounter* and a port interface type containing a method whose signature is *int tick(int)*. From now on, each time the name *ICounter* is used the type checker has access to its definition.

We now define component *Counter* providing an implementation of port interface *ICounter* at port *p*:

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

The type of component *Counter* is obtained from the type of the configurator expressions that define its internal component structure. In this case there are three configurators: *provides*, *methods* and *plug*. Each configurator is assigned a *script interface* and the type of the whole composition is obtained by composition of those smaller configurator types.

In particular, the type assigned to configurator *provides* is:

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

meaning that this kind of configurator must be composed with other configurators that satisfy the *open* resource (that introduces elements implementing port interface *ICounter* at a port named *p*). The information about the *provided* port of type *ICounter* is used to define the component type of all component definitions that use this configurator. Typing the method block configurator is more delicate since every unknown dependency must be discovered. In this case no problems arise since the configurator does not introduce any dependencies. This situation will show up later in the example.

The type assigned to configurator *methods* is:

```
script interface introduces {
    available {int tick(int)} m;
}
```

This type introduces an available resource featuring all implemented methods. This

information is used when connecting implementations to ports. Internally, *{int tick(int)}* is represented by an anonymous port interface type.

The last configurator used in the definition of component *Counter* is a *plug* configurator. The type assigned to this configurator expression is:

```
script interface needs {
    available Y m;
    open X p;
} introduces {
    available Y m;
}
```

The meaning of this type indicates that in order to perform a plug operation we must have an available resource named *m* and an open port named *p* with a compatible type.

Notice that this configurator is typed by itself and that there is no information on the types of *p* and *m*. That information is only available when the configurator is composed with others. To solve this problem we must abstract the types of the operands. Type variables *X* and *Y* are assigned to the resource name. They will later be checked and instantiated by the type inference mechanism explained in Chapter 5. Besides the type variables assigned to each resource name, a condition must be established between both type variables, stating that type *Y* must be a subtype of type *X*, i.e. the methods implemented by *m* must be, at least, the ones provided by port *p*. This condition must be true at all times. If once the type variables are solved this condition becomes false, then the type checker reports a typing error.

In what concerns the type information kept in the typing environment, for each component declaration a new scope is created in the environment. The association between names and types within a component scope actually depends on the configurator itself. Adding to the typing environment the *script interface* associated with each configurator is not necessary since they are only used in composition operations, and by that time their types are known. Also, since the association is based on names, the problem about what name to assign the type to would arise. The only reason to create a new scope in the typing environment for each component declaration is to deal with repeated names. As such, the association in the typing environment is between the name given to the declared resource and its own type. For instance, configurator:

```
provides ICounter p;
```

leads to the association between the name *p* and port interface type *ICounter*. For the method block declaration:

```
methods m {
    int s = 0;
    int tick(int n) {
        s = s + n;
        return s;
    }
};
```

the association in the typing environment is between name *m* and the anonymous port interface type featuring method *tick*. Since the *plug* configurator is not declaring any new name, no association is added to the typing environment in this case.

At this point all configurators used within the component definition can be composed together. When composing configurators, their types are also composed according to the rules established in Section 4.4.2.

For instance, taking the first two configurators, *provides* and *methods*, and composing their types we get the following type:

```
script interface introduces {
    open ICounter p;
    provided ICounter p;
    available {int tick(int)} m;
}
```

The new type is obtained by merging the two types together. This means that the composition of both configurators, besides providing a port that is still not connected to any implementation, also introduces a method block named *m* which is available to be connected to other ports.

Composing the new compound type with the one of configurator *plug* is a more delicate process. In this case there are needed resources that must be satisfied, however, the types of those resources are still undefined. Lets begin with the introduces resource:

```
available Y m;
```

This resource encounters no problems besides the fact type *Y* is still to be instantiated. Since another resource named *m* already exists in the other script interface type, one can replace type *Y* by the type of that resource. In this case the type is *{int tick(int)}*. As such, the resources can be added to the resulting type, becoming:

```
script interface introduces {
    open ICounter p;
    provided ICounter p;
    available {int tick(int)} m;
    available {int tick(int)} m;
}
```

Taking the needed resource:

```
available Y m;
```

we know that type *Y* is *{int tick(int)}*, the same as in the previous resource. When merging this resource with the resulting type we find that the same available resource *m* is introduced. This means that the needed resource is satisfied, and neither the introduced or the needed resources should be part of the final type. To avoid satisfying needed resources with the introduced resources of its own type, a local copy of the compound type is kept, and matches for compatible resources are searched in that copy (that remains unaltered throughout the whole composition). After this operation, the resulting type is:

```
script interface introduces {
    open ICounter p;
    provided ICounter p;
    available {int tick(int)} m;
}
```

Notice that only one of the introduced available resources is removed. This occurs because only one resource can satisfy another, and not many. The remaining needed resource in the type joining the composition is:

```
open X p;
```

Following the sames steps as before we conclude that *X* is of type *ICounter*, the same of the provided port *p*. Merging this resource with the resulting script interface type means that there will cease to exist an introduced open port, as that same port is needed by another configurator and, therefore, fulfilled. The type representing the composition of the three configurators that define component *Counter* is:

```
script interface introduces {
    provided ICounter p;
    available {int tick(int)} m;
}
```

When the composition of all configurators is completed, two extra verifications must be performed, ensuring that no introduced open resources exist and that no needed resources exist either.

The reason for these two verifications is that if an open resource remains as introduced it means that a provided port is still not connected to an implementation. If a component is created with such a feature, calling a method in the open port would lead to a *method-not-found* runtime error, thus defeating the purpose of the type system. The same thing happens if there are needed resources, since their existence can lead to runtime errors too.

In our example none of the two situations occurs, so the component can be created. The type assigned to the component is a *component interface* type, and will feature all the provided and required ports of the component:

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

The association between name *Counter* and the component interface type above is added to the typing environment at this point.

To better illustrate the typing of component definitions that depend on other components lets take the following definition of component *ZeroCounter* that requires an external service named *c* of type *ICounter* that is used in the implementation of the method block:

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

The *provides* and *plug* configurators are typed the same way as in the previous example. Configurator *methods*, however, now needs an external resource *c* where method *tick* can be called. This external requirement must be present in the script interface type associated with the configurator. Also, another configurator *requires* is introduced, meaning that an external implementation is needed in this component. By assigning types to *provides* and *methods* (the other two configurators remain the same), we have:

```
script interface introduces {
    available ICounter c;
    required ICounter c;
}
```

for the *requires* configurator, and:

```
script interface  needs {
    available W c;
} introduces {
    available {int tick(int)} mm;
    available W c;
}
```

for the method block configurator. Here, the needed and introduced available resource *c* refers to the external resource that must be fulfilled. Performing the step by step composition as before, we get that type variable *W* is instantiated by type *ICounter*, and the resulting type of the composition of all configurators is:

```
script interface introduces {
    provided ICounter p;
    required ICounter c;
    available ICounter c;
    available {int tick(int)} mm;
}
```

Performing the same verifications as before, to check if the component can be created, we find that there are no resources in the *needs* set, nor any *open* resources. This means that component *ZeroFilter* can be created, and will be assigned type:

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

Using both components (*ICounter* and *ZeroFilter*) to create new objects is performed by means of the *new* operation:

```
count = new Counter;
zf = new ZeroFilter with [c := count.p];
```

Since *object interface* types only keep track of provided ports, we find that the type of both objects *count* and *zf* is the same:

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

This occurs because, for a component to be instantiated, all required ports must be explicitly connected to some provided port of the same type. The connectivity is made throughout the *with* clause, and the type system ensures that the types between both ports are compatible. The notion of compatibility between types, in this case, means that a provided port $p$ can be connected to a required port $r$ if $p$ provides, at least, all methods required by $r$.

In the following sections deeper details on the implementation of both the static type checker and the dynamic verifier are given. Details about the type inference

mechanism in ComponentJ's type system, as well as specific examples showing its execution, are presented in Chapter 5.

## 4.4 Static Type Checker Implementation

Static type checking refers to the type verification process performed at compile time to ensure that a certain group of execution errors will not occur.

The type system presented here is based on the one presented on [28], and is implemented on top of the compiler introduced in [35]. The type evaluation process follows a set of pre-defined rules, and checks for type incompatibilities. The four types specific to ComponentJ's constructs are port interface, component interface, object interface and script interface.

Types are represented in the compiler by objects that implement a common interface CJType. The Java class CJPortInterfaceType is used to represent *port interface types*, and features both the set of methods available at the port (meths) and the name of the port they are associated to (name):

```
public class CJPortInterfaceType implements CJType {
    private String name;
    private List<CJMethodDeclaration> meths;
    ...
}
```

Class CJMethodDeclaration represents a method prototype with a return type, a name, and a list of parameters (each characterized by a name and a type).

*Component interface types* are represented in class CJComponentType, and contain the set of provided and required resources of a component. To make verifications easier by the type system, two Map data structures are used to represent the type, using the resource name as the key:

```
public class CJComponentType implements CJType {
    Map<String,CJType> required;
    Map<String,CJType> provided;
    ...
}
```

The same data structure is used to define *object interface types* in class CJObjectType, although objects only feature the provided resources:

```
public class CJObjectType implements CJType {
    Map<String,CJType> provided;
    ...
}
```

To represent *script interface types* more information is needed. The first separation of information that can be found is the two resource sets introduces and needs, which are represented in class CJScriptType as:

```
public class CJScriptType implements CJType {
    CJResources introduces;
    CJResources needs;
    ...
}
```

Class CJResources includes four different sets, one for each kind of resources that can exist in a script interface type (available, open, provided and required). Having a different set for each kind of resources makes it easier to perform some verifications (such as if no open resources exist). A single resource is only represented by its name and type because the information about its kind depends on the set the resource is added to.

The types assigned to each ComponentJ construct are presented next, along with an explanation of their typing rules.

### 4.4.1 Configurators

For the expressions denoting configurators a script interface type is created according to the information featuring the configurator in question. The types assigned to each configurator are as follows:

- **provides** $\tau$ **p:** Introduces a provided port of a previously defined port interface type $\tau$, as well as a new open resource also of type $\tau$ (i.e. a port that is not yet connected to any implementation):

```
script interface introduces {
    open τ p;
    provided τ p;
}
```

- **requires $\tau$ p:** Introduces a required port of a previously defined port interface type $\tau$. The required resource also becomes available, so an available resource, also of type $\tau$, is introduced too:

```
script interface introduces {
    available τ p;
    required τ p;
}
```

- **uses c = Comp:** This operation is used to introduce an inner component. As such, its type will feature the structure of component *Comp*. The resulting type will show an introduced available resource for every port provided by *Comp*, and also an introduced open resource for every required port of *Comp*. This ensures that, before the outer component is instantiated, all requirements of the inner components must be fulfilled:

```
script interface introduces {
    available Comp.getProvided1;
    ...
    available Comp.getPovided_n;
    open Comp.getRequired1;
    ...
    open Comp.getRequired_n;
}
```

- **methods m {$m_1$($p_1$,...,$p_n$); ... ; $m_n$($p_1$,...,$p_n$)}:** The type assigned to the definition of a new method block must ensure two properties: first, the methods must become available (by means of an introduced available resource), and also, all ports or inner components accessed inside the method definition must be available whenever the method block is connected to a port. This is performed by adding a needed available resource for every port or component required by methods $m_1$

to $m_n$. These requirements are also introduced as available resources that might be used in another configurator if need be:

```
script interface needs {
    available m1.getRequirement1;
    ...
    available m1.getRequirementn;
    ...
    available mn.getRequirement1;
    ...
    available mn.getRequirementn;
} introduces {
    available {m1(p1,...,pn); ... ; mn(p1,...,pn)} m;
    available m1.getRequirement1;
    ...
    available m1.getRequirementn;
    ...
    available mn.getRequirement1;
    ...
    available mn.getRequirementn;
}
```

- **plug m into p:** The plug configurator is used to connect an implementation to a specific port. If $m$ is available, and $p$ is an open port then they can be connected, making $m$ available. This is expressed in the configurator type as two needed resources: one open and one available, as well as an introduced available resource $m$. The specific types of both $m$ and $p$ are not yet available at this time, and will be inferred when a composition operation occurs.

```
script interface needs {
    available X m;
    open Y p;
} introduces {
    available X m;
}
```

### 4.4.2 Composition Operation

When two configurators are composed together, the resulting type of this operation is also a composition of their types. Take two script interface types $\tau_1$ and $\tau_2$, and the

```
CJScriptType compose(CJScriptType type1, CJScriptType type2) {
    Unifier uni;
    Resources needs1 = type1.getNeeds();
    Resources intro1 = type1.getIntroduces();
    Resources needs2 = type2.getNeeds();
    Resources intro2 = type2.getIntroduces();

    for (Resource r1 : intro1) {
        Resource r2 = needs2.get(r1.getName());
        r1.unify(r2, uni);
        if (r1.satisfies(r2))
            intro1.remove(r1);
        else
            needs1.add(r2);
    }
    for (Resource r : intro2)
        intro1.add(r);
}
```

**Figure 4.7** Algorithm for the composition operation between configurator types

composition operation between them $\tau_1;\tau_2$. When composing these two types the introduced resources of $\tau_1$ might match some needed resources of $\tau_2$. If a match occurs it means that when composing the two configurators ones requirements become fulfilled, and that information is reflected in the type by removing the matching resources. All resources that are not matched are merged into the resulting type. The algorithm used to perform a composition between two script interface types, Figure 4.7, shows how the operation is performed.

Since some resources might still not have a specific type assigned to them (as is the case in the plug configurator type), when composing two script interface types together the missing types are inferred. This is represented by the *unify* operation in Figure 4.7. The *unify* operation checks for any equation that must be added to the type inference mechanism and, if some equation is added, it solves the system of equations in order to instantiate all possible type variables. Equations that must be added to the type inference mechanism are the ones that feature, at least, one type variable. For instance, if we take the following two resources:

```
available X p;

available I p;
```

where $X$ is a type variable and $I$ a previously defined port interface type. If this situation occured during a composition operation, a new equation would be created stating that type variable $X$ has the same value as port interface $I$.

### 4.4.3 Components

Compose expressions and component declarations are assigned component interface types after the evaluation process. Since all components are defined by means of configurators composed together, their type will be based on the type resulting of the composition. Not all configurators can be used to define a component since some properties must be ensure for the system to behave well. Take a script interface type $\tau$ denoting the configurator used to define the component. Two conditions must be met before the component can be created:

- All needs must be satisfied, i.e. no resources can exist in the needs set of $\tau$;
- No open ports can be introduced, i.e. no open resources can exist in the introduces set of $\tau$.

Having these rules ensures that no components are created if they provide, for instance, ports with no implementation, or if the needed resources for a plug operation do not exist. If both conditions are true then the component can be created and its type will be a component interface type featuring all provided and required resources present in $\tau$.

### 4.4.4 Objects

Objects can be instantiated from any previously created component using the *new* expression. Since components might introduce some requirements (also represented in the component type itself), when an instantiation takes place those requirements must be met through the with clause. If an assignment is present for every required port, then the object can be created. Since all requirements have been fulfilled, the object is

```
boolean checkObjectInstantiation(CJComponentType type, List<Port> assigns){
    int numReq = 0;
    for (Port p : type.getRequiredPorts) {
        if (!assigns.contains(p))
            return false;
        numReq++;
    }
    return (numReq != assigns.size()) ? false : true;
}
```

**Figure 4.8** Object instantiation verification algorithm

typed with an object interface type featuring only the provided ports of the component it is instantiated from. The algorithm used to perform the verification of an object instantiation is described in figure 4.8, and ensures that all required ports are assigned through the *with* clause of *new* expression.

## 4.5 Dynamic Type Verification

In opposition to static verification, dynamic type checking is performed at runtime, and is defined in [14] as being *"a collection of runtime tests aimed at detecting and preventing forbidden errors"*, i.e. it ensures that operations performed on the running system are checked for correctness before being applied. This kind of type checking allows programming languages to introduce some operations that would otherwise be rejected by a static type checker (such as dynamic reconfiguration in ComponentJ).

Reconfiguration operations in ComponentJ allow changes to both the architecture of the system and its implementation. Since this operation is based on runtime information only some aspects of the operation can be verified at compile time, the rest of the verification being delayed to runtime. The verification relies on a runtime test that checks if the given configurator is applicable to its target object before its application. The test is based on the inspection of the object's structure and on a limited amount of runtime type information on both configurators and objects.

For the dynamic type checking procedure to take place, we need to store runtime type information within the structure of objects and reconfigurators. As such, objects

are represented by their required ports, inner elements and provided ports. Configurator values consist of a runtime representation of a sequence of instructions to construct or change the internal structure of an object, and a script interface type specifying the precondition for its application (the *needs* set of a script interface type). The runtime support system developed in [35] already includes the necessary information about objects in class CJObject:

```
public class CJObject {
    Map<String, PortProvider> requires ;
    Map<String, PortProvider> provides ;
    Map<String, CJObject> uses ;
    ...
}
```

where the *requires* and *provides* Hashtables represent the required and provided ports of the object, and the *uses* Hashtable represents inner elements of the object.

Reconfigurators (i.e. configurators used to reconfigure objects) are represented by the runtime system through class Reconfigurator:

```
public class Reconfigurator {
    private IConfigurator configurator;
    ...
}
```

where IConfigurator is the common interface implemented by all configurators:

```
public interface IConfigurator {
    void perform(CJObject o);
    boolean test(CJObject orig, CJObject c);
}
```

Method *perform* is used to apply the configurator to object *o*, and method *test* determines if the reconfiguration operation can be safely performed. A class implementing interface IConfigurator exists for every configurator of the language, as well as for the composition operation between configurators. The original definition of these the configurator classes and the reconfigurator one featured no type information. However,

```
public boolean match(CJObject o, CJResources pre) {
    for (Resource provRes : pre.getProvided())
        if (!o.isInProvided(provRes)) return false;
    for (Resource reqRes : pre.getRequired())
        if (!o.isInRequired(reqRes)) return false;
    for (Resource availRes : pre.getAvailable())
        if (availRes.hasCompoundName())
            if (!o.isInInnerProvided(availRes))
                return false;
        else
            if (!(o.isInRequired(availRes) || o.isInInnerAvailable(availRes)))
                return false;
    for (Resource openRes : pre.getOpen())
        if (openRes.hasCompoundName())
            if (!o.isInInnerRequired(openRes)) return false;
        else if (!o.isInProvided(openRes)) return false;
    return true;
}
```

**Figure 4.9** Algorithm for the runtime check of reconfiguration operations

for the type checker to run properly, that type information is added in the form of a new field representing the type of the configurator/reconfigurator.

The dynamic verification of a reconfiguration operation is based on a matching test, formalized in [28], stating that for a given object *o* and a configurator type *c*, each resource found in *c* can be found, with compatible types, in one of the records of *o* (provided ports, required ports or inner elements) according to the following:

- Resources representing provided ports in *c* must be checked in the provided ports set of object *o*;
- Resources representing required ports in *c* must be checked in the required ports set of object *o*;
- Available resources in *c* must be checked either in the required ports or the inner elements set of object *o*;
- Available resources with compound names (e.g. x.l) in *c* can only be related to provided ports of inner elements of object *o*;

- Open resources with simple names in *c* can only refer to provided ports of object *o*;
- Open resources with compound names in *c* can only be required ports of inner component of object *o*;

If the matching test succeeds it means that configurator and object are compatible, and that the reconfiguration operation can safely be executed. If it fails, then the reconfiguration is not performed, and an alternate path of execution is taken (defined in the else branch of the reconfiguration declaration). The method in class Reconfigurator used to support the reconfiguration operations is defined in figure 4.9. Notice that resources have type information, and checking if a certain resource exists within a record takes n account not only the resource name but also its type.

# Chapter 5
# Type Inference in ComponentJ

***Infer:*** *to form an opinion or decide that something is probably true because of other information you already know.*

When dealing with types in programming languages there is always one question that comes to mind: why should the programmer have to specify all the type information? Why can't the compiler guess? In fact, in programming languages such as ML, the compiler is able to deduce the necessary type information so that the programmer does not have to introduce explicit type annotations in the source code. These kind of programming languages are said to have implicit typing, in opposition to explicitly typed languages, where type annotations are used to help guide the type checker. Having type inference mechanisms, which allow implicit typing, available within a compiler restricts possible runtime errors due to typing mistakes, as it does not allow the programmer to introduce wrong type information, and, in some cases, prevents the need for repeated type information. One of the pioneer algorithms to perform type inference is described in [24], known as the Damas-Milner algorithm, or algorithm W, which is used to find the most suitable type for any expression, and which has been proved to be both sound and complete [17].

Taking ComponentJ's original calculus, both method block declarations and plug operations had to be annotated with type information. As these type annotations can be very long and complex, the annotating job becomes cumbersome. In this work, one of the goals is to prevent the need for those type annotations, by equipping the type system with type inference mechanisms that make the language implicitly typed. The type inference mechanism used is based on algorithm W, explained next.

## 5.1 Algorithm W

The main idea behind algorithm W is that every language construct that needs its type inferred is associated with a set of constraints that define its type. The type inference mechanism will then take all constraints and create with them a system of constraints that can be seen much as a system of equations in the mathematical sense. In order to solve the system of constraints an unification algorithm is performed to find a suitable set of substitutions. If at some point in the algorithm the system has no solutions then there is a typing error in the program. If the algorithm finishes, then the solution represent the wanted types.

## 5.2 The Unification Algorithm

Unification is mostly used in first-order logic, being the basis of most work in automated deduction, and of the use of logical inference in artificial intelligence. A *unifier* is defined as being a substitution that makes two terms identical. The unification algorithm takes as input two terms *t1* and *t2*, and tries to find a unifier for them, i.e, a substitution that makes both terms equal. To do so, terms equality is expressed by means of equations, and all equations are stored in a stack $E$. There is also a substitution set $S$ used to store the substitutions obtained, and that originally is empty.

As it can be found in several logic and artificial intelligence books and papers, being its origin [27], the algorithm takes for input two terms $t_1$ and $t_2$ to be unified, and its output is either failure, if at some point it is forced to stop, or the location $S$, containing the substitutions obtained. The algorithm starts with $S$ empty and $E$ to contain all equations $t_1 = t_2$. After that, and while $E$ is not empty, pop $X = Y$ from $E$ and applies one of these cases:

1. If $X$ is a variable that does not occur in $Y$: substitute $X$ for $Y$ in $E$ and in $S$; add $X = Y$ to $S$

2. If $Y$ is a variable that does not occur in $X$: substitute $Y$ for $X$ in $E$ and in $S$; add $Y = X$ to $S$

3. If $X$ and $Y$ are identical constants or variables: do nothing

4. If $X$ is $f(X_1,...,X_n)$ and $Y$ is $f(Y_1,...,Y_n)$ for some functor $f$ and $n > 0$: push $X_i = Y_i$, $i = 1...n$ on $E$

5. Else output is failure

The next example shows an execution of this algorithm, to find an unification between terms

```
T1 = append ([a,b], [c,d], Ls)
T2 = append ([X|Xs], Ys, [X|Zs])
```

The first step in the execution, as seen before, is the initialization of both $E$ with the equations to be solved, and $S$ empty:

*Equations E:*

```
append ([a,b], [c,d], Ls) = append ([X|Xs], Ys, [X|Zs])
```

*Substitution S:* { }

After that step has been taken, the equation is popped from $E$ and processed. In this case we have the same functor in both $X$ and $Y$, so rule (4) is applied. By doing do, the status of both sets is:

*Equations E:*

```
[a,b] = [X|Xs]
[c,d] = Ys
Ls = [X|Zs]
```

*Substitution S:* { }

The next equation to be popped is `[a,b] = [X|Xs]`. In this case, the functor is also the same in both $X$ and $Y$, so the same rule as before is applied, resulting in:

*Equations E:*

```
a = X
[b] = Xs
[c,d] = Ys
Ls = [X|Zs]
```

*Substitution S:* { }

For the next two equations to be popped from $E$, rule (2) is applied since $X$ is a variable that does not occur in constant $a$. After the two equations are popped from $E$ and processed, and all substitutions are added to $S$, we obtain:

*Equations E:*

```
[c,d] = Ys
Ls = [a|Zs]
```

*Substitution S:*

```
X = a
Xs = [b]
```

The next equation in line is treated with rule (3), since the variable is on the right side of the equation. The last equation on $E$, `Ls = [a|Zs]` uses rule (2), as used previously. The result, after processing the last two equations is:

*Equations E:* { }

*Substitution S:*

```
X = a
Xs = [b]
Ys = [c,d]
Ls = [a|Zs]
```

As there are no more equations in $E$, and no failure was reported, then the algorithm terminates successfully and returns the substitutions $S$. At this point, $E$ is empty, so the algorithm returns the substitution $S$.

## 5.3 Type inference in ComponentJ

In the context of programming languages, the unification algorithm is executed the same way as it is in logic. The difference is that equations, in this case, relate to type information instead of assertions. The two possible outcomes of the algorithm are failure, in which case there is a typing error, or success, where the resulting substitution representing the needed type information is returned, allowing the type system to reach a conclusion.

The main challenge in using this technique in programming languages is making sure the equations are generated correctly. In what concerns the ComponentJ programming language type inference is only provided for plug operations, method calling and port accessing. The type of a plug operation is a configurator type so the eventually needed type information is provided when composing its type with the one of other configurators. The same does not occur in method call and port access. In this cases the type of the expression can be undefined, since it represents the return type of the method being called. The only time these expressions have an undefined type is when it is being used in the definition of a method block, since by that time the port/object might have not been created yet. Introducing type variables in this context is reflected in the type of the method block definition the operations are used, so, once again, whenever the method block is composed with other configurators, the unification algorithm is executed and substitutions for the type variables are found.

The introduction of type variables in the type system can not be *naive* in the sense that some properties must be ensured between type variables so that everything goes right. These properties are represented by means of conditions that must be true at all times during the type verification. If a condition includes type variables then it is assumed to be satisfied, however, once type variables are replaced by some specific types, the conditions must still hold. Since the only time where type variables might be replaced by specific types is during a composition operation, conditions are only checked at this moment too. The conditions introduced by each of these operations is presented next, as well as the composition operation and how it is performed.

**Plug operation:** plug m into p

In this case the type assigned to the configurator is

```
script interface introduces {
    available X m;
} needs {
    available X m;
    open Y p;
}
```

Eventhough $X$ and $Y$ are still unknown types at this moment, it is known that type $X$ must be a subtype of $Y$, since the method block $m$ being connected to port $p$ must contain an implementation of all methods available in $p$. This relation is expressed by the following condition:

```
X <: Y
```

**Method call:** p.m($t_1$, ..., $t_N$)

In what concerns method call expressions, its type is the return type of method $m$. Inside a method block definition this type is set to a type variable $X$. The only known information is that method $m$ must be one to the methods available in port $p$ once port $p$ is defined. Assuming the type of port $p$ is $Y$, this relation is described in the following condition:

```
Y <: port interface { X m(t1, ..., tN) }
```

Notice that parameter types of the method might also be still undefined. If it is the case it means that the return value of a method call is being used as a parameter, so this same rule (or the one of port accesses) applies.

**Port access:** o.p.m($t_1$, ..., $t_N$)

A port access operation is a special case of a method call expression. In our case it is treated the same way as a method call (the same condition is introduced) with the only difference being that its reflexion on the type of the method block definition the introduced/needed resources regarding this operation have a compound name

*o.p*. This allows the verification that port *p* exists in object *o*. The type assigned to this expression is also the return type of method *m*.

If either method call or port access operations are used in an expression (e.g. in a sum) then new conditions can be introduced (e.g. the return type must be a numeric type).

**Composition Operation:** $\mathsf{conf}_1 \; ; \mathsf{conf}_2$

When composing two configurator with each other, their types must be composed as well. Each configurator has a script interface type asssigned to it, and the main idea is that the resources in the *introduces* set of one type will be used to fulfil resources present in the *needs* set of the other type. For one resource to fulfil another three conditions must be ensured:

1. Resources are of the same kind (available, open, provided, needed);

2. Resources have the same identifier;

3. Resources have compatible types;

If any of this conditions fails resources remain unfulfilled. The first two conditions are easily checked since each resource is assigned a specific kind and identifier when created. However, as it was said before, the type of the resource can represent a type variable. In this case it is not possible to check directly if types on both resources are compatible. Everytime this occurs, i.e. a comparison between types is not possible because one of the types is undefined, one equation is added to the type inference mechanism. The equation represents the assignment between the type variable of one resource and the type of the other one. It is assumed that if the resources have the same name and kind, they either fulfill each other or there is a typing error, so, when this situation occurs, the resources are assumed to be satisfied until the unification algorithm is executed.

After the composition is concluded, one of two outcomes is possible: there are no equations ins the type inference mechanism, in which case the composition operation

returns; or equations have been added to the type inference mechanism, and the unification algorithm is performed. If the algorithm finishes successfully then the composition operation might return. If, however, the algorithm fails, it means that at least two resources with the same kind and identifier do not have compatible types in which case a typing error is returned.

If the execution reaches this point without failing, conditions, if any, must be checked. The type variables present in conditions must also be replaced by the values found through the unification process. If any of the conditions is false a typing error is also to be reported. It is only after this step that the composition operation can return successfully.

## 5.4 Short Example using Type Inference in ComponentJ

The next example shows how type inference is performed. In this case only plug operations and method calls are used. Consider the following example:

```
port interface I {
    void set(int i);
}

port interface J {
    int get();
}



c1 = provides I p;
c2 = requires J r;
c3 = methods m {
       int i = 0;
       void set (int a) {
         i = r.get() + a;
       }
     };
c4 = plug m into p;
c5 = c1; c2; c3; c4;
```

The types of configurators $c_1$ to $c_4$ are the following:

```
c1 : script interface introduces {
         open I p;
         provided I p;
     }
c2 : script interface introduces {
         available J r;
         required J r;
     }
c3 : script interface introduces {
         available {void set(int)} m;
         available X r;
     } needs {
         available X r;
     }
c4 : script interface introduces {
         available Y m;
     } needs {
         available Y m;
         open Z p;
     }
```

Types $c_3$ and $c_4$ contain type variables, as well as type conditions that must be true at all times. In this case, conditions are:

```
X <: port interface (W get())
W.isNumerical()
Y <: Z
```

where $W$ represents the undefined type of port $r$. The second condition is introduced since the return value of the method call operation is used in a plus operation (that by definition requires two numerical values).

The existence of type variables is not a problem yet, since configurators are not being applied anywhere. The compose operation, however, requires all configurators to be composed with each other, and in that case, type variables must be assigned a value. When composing $c_1$ with $c_2$, since both types have only introduced resources, the following type is obtained:

```
c1;c2 : script interface introduces {
          open I p;
          available J r;
          provided I p;
          required J r;
        }
```

To compose the resulting type of $c_1$ ; $c_2$ with $c_3$ the procedure is the one described before. When comparing the needed resources from the type of $c_3$ with the introduces resources of the type of $c_1$ ;$c_2$ we come across resources of the same kind and with the same name, but where one of the types is still unknown. This occurs with resources available X r and available J r. Since $X$ represents a type variable, the equation X = J is added to the type inference system. No other resources can be fulfilled, so the next step is to solve the system of equations created. In this case it consists only of one equation assigning a type variable to a specific type, so the substitution to be made is X = J. The substitution will be applied to every resource of the composed type that contains a type variable $X$, as well as to all conditions the type may have. After the substitution is applied, the conditions are the following:

```
J <: port interface (W get())
W.isNumerical()
Y <: Z
```

The second condition still contains type variables, so it is automatically checked as true. The first one, however, represents a relation between two port interface types (since $J$ is a port interface type itself). The method represented in the supertype still has an undetermined return type. In this case one of the following must occur: no method named *get* with 0 arguments exists in $J$, in which case the condition is not satisfied leading to a typing error; or there is a method *get()* in $J$ with a return type $\tau$. In this last case, the type variable $W$ is replaced by $\tau$ and either at least one condition fails leading to a typing error, or both conditions are checked true and the verification succeeds. The resulting type of the composition operation between $c_1$, $c_2$ and $c_3$ is:

```
c1;c2;c3 : script interface introduces {
             open I p;
             provided I p;
             required J r;
             available J r;
             available {void set(int)} m;
           }
```

Performing the same steps with the composition of configurator $c_1;c_2;c_3$ with $c_4$ we obtain the following equations for the unification algorithm:

```
Y = {void set(int)}
Z = I
```

and also the condition:

```
Y <: Z
```

In this case the system of equations obtained is also very simple, and consists only on the replacement of a type variable with a specific type. Performing the substitutions of $Y$ and $Z$ on the type expressions and conditions, we obtain the following condition:

```
port interface {void set(int)} <: I
```

Checking the subtyping relation results in a valid condition. This means the composition operation is type safe, and its resulting type is given by:

```
c1;c2;c3;c4 : script interface introduces {
                provided I p;
                required J r;
                available J r;
                available {void set(int)} m;
              }
```

# Chapter 6
# Subtyping Relation Between Configurators

***subtype:*** *a special type included within a more general type.*

In chapter 4, a type was defined as being a set of elements having specific characteristics. All elements inside one of these sets must satisfy all the set's properties, however, no problems should arise if an element satisfies not only the sets properties, but more. This is the intuitive notion of subtype: a value that fits in one type, but that has more features.

The use of subtyping relations increases the reliability of systems, as it provides a way of increasing functionality without having to change the original system [13]. The most common way to introduce subtypes in object-oriented programming languages is by using inheritance, where a sub-class takes on all the features of its super class, and increases its functionality with its own specific features. In the typing process, subtyping introduces a new principle of substitution: whenever an object of a type is expected, an object of one of its subtypes can be used instead, and whenever a subtype is expected, neither the supertype nor another subtype can be used. If these rules are followed, subtyping relations can be safely used to make substitutions more flexible.

Within ComponentJ's type system, subtyping relations between port interface types, object interface types and component interface types have already been described in chapter 4. These subtyping relations are as intuitive as the one present in object-oriented programming languages. However, script interface types, representing the type of configurators, have a more complex structure and, therefore, defining a subtyping relation between them is not as intuitive. The naive approach is to assume all script types are alike, and treat them similarly to component interface types. If this approach is taken, a subtyping relations between two script interface types $\tau_1$ and $\tau_2$ ($\tau_1 <: \tau_2$) exists if $\tau_1$ introduces at least and needs at most, the same resources as $\tau_2$.

Although it might seem like a plausible approach, a simple example suffices to show it is not a possible relation.

Take configurators plug m into p and methods m { ... } typed with $\tau_1$ and $\tau_2$ as follows:

```
τ1 = script interface introduces {
        available | m;
     } needs {
        available | m;
        open | p;
     }

τ2 = script interface introduces {
        available | m;
     } needs { }
```

Taking this approach a subtyping relation between $\tau_1$ and $\tau_2$ could be established as $\tau_1 <: \tau_2$. This would mean that, in what concerns the type system, replacing a plug operation by a method block would be acceptable since a method block, in this case, would introduce the same resources and would not need any other. However, intuitively a method block definition is not a *"sub-configurator"* of plug, since the semantics of both configurators is different, thus, their types cannot be related by a subtyping relation.

The structure of a configurator type is based on resources of different kinds. Since different kinds represent different properties of the type, different kinds of resources should not be related. This means that each kind of resource must be treated separately. The subtyping relation between two configurators should then be established if all their resources can be related by means of subtyping relations. For each kind of resources, two aspects must be taken into account in the definition of a subtyping relation: first, the sets of resources in both types, where, for there to be a possibility of a subtyping relation, one of the sets must be a subset of the other, and second, the type of each matching resource (resources of the same kind with the same name) must allow a subtyping relation also.

For the sake of simplicity, in this chapter we use the notation of [28] to represent script interface types. A script interface type is represented as K $\Longrightarrow$ K' where K represents the needs set and K' the introduced set. To represent resource kinds the following notation is used:

$$\begin{aligned}
&\textbf{available}\ \tau\ \mathsf{p} = \mathsf{p} \bullet \tau \\
&\textbf{open}\ \tau\ \mathsf{p} \quad\ = \mathsf{p} \circ \tau \\
&\textbf{provided}\ \tau\ \mathsf{p} = \mathsf{p} \rhd \tau \\
&\textbf{required}\ \tau\ \mathsf{p} = \mathsf{p} \lhd \tau
\end{aligned}$$

Using this notation the type of the plug operation is written:

$$\{\mathsf{p} \circ \tau_1\,,\ \mathsf{m} \bullet \tau_2\} \implies \{\mathsf{m} \bullet \tau_2\}$$

To identify a specific resource in a set of resources we write:

$$\mathsf{K}, p \circ \tau_1 \implies \mathsf{K'}$$

The relation between the same kind of resources in both needs and introduced sets is presented next. The base principle taken into account is that in a subtyping relation, a subtype can always replace the supertype without introducing runtime errors.

## 6.1 Provided and Required resources

Introduced provided and required ports are the easiest to understand since they relate the same way as provided and required ports in component types. Adding a subtype with more provided ports where a supertype is expected is not a problem, since it only indicates that new ports exist but will not be used (only the ones of the supertype are used). In what concerns required ports, having less required ports in the subtype is also not an issue, since it only represents that not all resources needed by the supertype will be used. These two rules can be represented as:

$$\frac{K \implies K' <: L \implies L'}{K \implies K', p \rhd \tau <: L \implies L'} \tag{6.1}$$

$$\frac{K \implies K' <: L \implies L'}{K \implies K' <: L \implies L', p \lhd \tau} \tag{6.2}$$

These rules relate the overall provided and required resources. They do not, however, represent how the resource types are related. In the case of provided resources,

if replacing the supertype port by the subtype one, the subtype must include, at least, the same methods as the supertype. Extra methods will be available but never used. On the other hand, when dealing with required resources, the opposite occurs, i.e. the subtype port can have, at most, the same methods as its supertype. These two rules follow standard covariant and contravariant subtyping relations and are represented as:

$$\frac{\tau \ <: \ \tau'}{K \Longrightarrow K', p \triangleright \tau \ <: \ L \Longrightarrow L', p \triangleright \tau'} \tag{6.3}$$

$$\frac{\tau' \ <: \ \tau}{K \Longrightarrow K', p \triangleleft \tau \ <: \ L \Longrightarrow L', p \triangleleft \tau'} \tag{6.4}$$

## 6.2 Available resources

Available resources represent resources that are ready to use, i.e. that can be connected to other elements (for instance a method block or internal component). When these kind of resources exist in the introduces set of a script interface type of a configurator $c$ it means that the resource can be used by other configurators, while if it is present in the needs set it means that the resource must be made available by some other configurator so that $c$ can make use of it.

Taking available resources in the needs set of the type first, we know that the configurator needs a certain set of methods described by the resource. Replacing such a configurator without introducing errors means that no more needed resources must exist, i.e. the configurator used to replace it can have, at most, the same needed available resources. This establishes the subtyping relation:

$$\frac{K \Longrightarrow K' \ <: \ L \Longrightarrow L'}{K \Longrightarrow K' \ <: \ L, p \bullet \tau \Longrightarrow L'} \tag{6.5}$$

Meaning that for a subtyping relation to hold, the supertype can have more needed available resources than the subtype. The relation between types of the same resource

can be expressed by:

$$\frac{\tau \ <: \ \tau'}{K, p \bullet \tau' \Longrightarrow K' \ <: \ L, p \bullet \tau \Longrightarrow L'} \tag{6.6}$$

If, however, the available resources exist in the introduces set of the type, they represent resources that can be used by other configurators. This means that, to replace a configurator $c$ that makes available a certain set of resources by another configurator $c'$, then $c'$ must make available at least the same resources as $c$:

$$\frac{K \Longrightarrow K' \ <: \ L \Longrightarrow L'}{K \Longrightarrow K', p \bullet \tau \ <: \ L \Longrightarrow L'} \tag{6.7}$$

The same principle applies if the same resource exists in two configurator types but with a different type assigned. In this case we get:

$$\frac{\tau \ <: \ \tau'}{K \Longrightarrow K', p \bullet \tau \ <: \ L \Longrightarrow L', p \bullet \tau'} \tag{6.8}$$

## 6.3 Open resources

Open resources are the more problematic ones, since they represent resources that are not yet satisfied. To establish a subtyping relation between open resources in the needs set, either the subtype features at least the same resources or at most the same resources as the supertype. Taking the first case, where the subtype features more open resources in the needs set, it is easy to realize that this relation is not possible. A configurator that requires some open resource $p$ cannot be replaced by one that also requires another open resource $p'$ without compromising the architecture, since resource $p'$ might not be available for connection. On the other hand, if the supertype needs more open resources than the subtype, one might say that no problems occur. However, in ComponentJ open resources are only needed by the plug operation so, replacing a supertype with more needed resources by a subtype with less would mean that provided ports might become unconnected to an implementation. Since none of this cases reflects the

possibility of a subtyping relation, configurators with different sets of open resources cannot be related through subtyping.

A similar outcome occurs when relating two configurators $c$ and $c'$ with needed open resources $r$ of type $\tau$ and $r'$ of type $\tau'$ accordingly, and that represent a subtyping relation $\tau <: \tau'$ between them. The two possible results of a subtyping relation are that either $c <: c'$ or $c' <: c$. Taking the first case and replacing the supertype $c'$ by its subtype $c$ we get that the resource needs even more methods. If the replacement takes place, the plug operation might not succeed anymore since the method block might not provide an implementation for the new methods. If, on the other hand, we assume the relation between the configurators to be $c' <: c$ the opposite problem arises, since if a certain port $p$ is provided with a set of methods available, if replacing the port in the plug operation the type of port $p$ by one that requires less method implementations, not all methods provided will have implementations connected to them.

Open resources are introduced in configurator types by means of configurators declaring new provided ports or denoting inner components. The same way no possible relation could be made between needed open ports, none can be defined between introduced ones since having a subtype introducing more open resources would require more connections to it, thus not satisfying the relation property that the supertype can always be replaced by the subtype. If, on the other hand, less open ports were introduced in the subtype then there could be connections to ports that no longer existed.

This allows the conclusion that, if two script interface types need or introduce a different set of open resources then they cannot be related by a subtyping relation, since such a relation would compromise the well functioning of the system.

## 6.4 Validation of the subtyping relation

Providing a formal proof for the subtyping relation between configurator type is beyond the goals of this dissertation. To prove the correctness of the relation a validation based on examples is provided in this section.

Configurators are mostly used to define components, so the first examples will show a correct component definition along with the same component definition with

some modifications to fit the various subtyping rules. In every case the component creation must still be possible, and eventual connections to the component cannot become faulty. First, take port interface definitions:

```
port interface I {
    void set(int i);
}
```

```
port interface J {
      void set(int i);
  int get();
}
```

These port interfaces are used to define the type of each resource. To define a component providing a port $p$ of type $I$ we have a set *conf* of configurator composed together (the type of each configurator is presented in front of each one):

```
component c {
    provides I p;      {} ⟹ {p ∘ I, p ▷ I}
    methods m {        {} ⟹ {m • {void set(int)}}
        void set(int i) {
        }
    };
    plug m into p;     {m •{void set(int)}, p ∘ I}⟹{m •{void set(int)}}
}
```

By composing the three configurator types we obtain the type $\tau$:

$$\{\} \Longrightarrow \{m \bullet \{void\ set(int)\}, p \triangleright I\}$$

meaning that the component the component can be created, and that it will provide a port $p$ of type $I$ to its outside.

The subtyping relation between components states that a component can be replaced by another if it provides at least and requires at most the same ports. If a new component $c'$ is defined using configurator *conf* composed with the following configurators (originating configurator *conf'*):

```
provides J q;      {} ⟹ {q ∘ J, q ▷ J}

methods m1 {       {} ⟹ {m1 • {void set(int), int get()}}
    void set(int i) {
    }
    int get() {
    }
};

plug m1 into q;   {m1 • {void set(int),int get()}, q ∘ J}
                                  ⟹ {m1 • {void set(int),int get()}}
```

Type $\tau'$, the type resulting of the compostion of the previous configurators with the new ones is:

$$\{\} \Longrightarrow \{p \triangleright I, q \triangleright J, m \bullet \{void\ set(int)\}, m_1 \bullet \{void\ set(int),\ int\ get()\}\}$$

Both configurators can be used to define components, and also component $c'$ is a subcomponent of $c$. In this case, the types of configurators *conf* and *conf'* should also be related by a subtyping relation such that $\tau' <: \tau$.

Taking the subtyping rules regarding provided resources and available resources, for the subtyping relation to hold both rules must be true. To simplify the representation take:

$$P = \{p \triangleright I, q \triangleright J, m \bullet \{void\ set(int)\}, m_1 \bullet \{void\ set(int),\ int\ get()\}\}$$
$$P' = \{p \triangleright I, m \bullet \{void\ set(int)\}, m_1 \bullet \{void\ set(int),\ int\ get()\}\}$$
$$P'' = \{p \triangleright I, m \bullet \{void\ set(int)\}\}$$

Through the direct application of rules 6.1 and 6.7 we get the following derivation:

$$\frac{\dfrac{\{p \triangleright I, m \bullet \{void\ set(int)\}\} <: \{\} \Longrightarrow \{p \triangleright I,\ m \bullet \{void\ set(int)\}\}}{P' <: \{\} \Longrightarrow \{p \triangleright I, m \bullet \{void\ set(int)\}\}}}{P <: \{\} \Longrightarrow \{p \triangleright I,\ m \bullet \{void\ set(int)\}\}}$$

Since a type is always subtype of itself, the derivation is true thus ensuring the subtyping relation between $\tau$ and $\tau'$ using two of the defined rules.

If instead of creating a new component with a new provided port we create a new

component $c''$ with a new required port:

```
component c'' {
    provides I p;     {} ⟹ {p ○ I, p ▷ I}
    required J q;     {} ⟹ {q ● I, q ◁ J}
    methods m {       {} ⟹ {m ● {void set(int)}}
        void set(int i) {
        }
    };
    plug m into p;    {m ●{void set(int)}, p ○ I}⟹{m ●{void set(int)}}
}
```

In this case, and again according to the subtyping relation between components, component $c''$ is a sub component of $c$. This establishes that a subtyping relation between the type resulting of the composition operations inside $c''$ ($conf''$) and type $conf$ such that $conf'' <: conf$ might exist. Configurator type $conf''$ is defined as:

$$\{\} \Longrightarrow \{p \triangleright I, q \triangleleft J, m \bullet \{void\ set(int)\}, q \bullet J\}$$

In this case the derivation obtained by the rules defined in this chapter is the following:

$$\frac{\{\} \Longrightarrow \{p \circ I,\ m \bullet \{void\ set(int)\}\}\ <:\ \{\} \Longrightarrow \{p \triangleright I, m \bullet \{void\ set(int)\}, q \bullet J\}}{\{\} \Longrightarrow \{p \circ I,\ m \bullet \{void\ set(int)\}\}\ <:\ \{\} \Longrightarrow \{p \triangleright I, q \triangleleft J, m \bullet \{void\ set(int)\}, q \bullet J\}}$$

From here on no rule can be applied, which means that the subtyping relation cannot be established. At a first glance it might seem like the wrong result, however, taking a deeper look, it is not possible to replace a configurator that introduces certain resources by one that doesn't. Replacing it could lead, for instance, to plugs with undefined resources.

The same occurs when dealing with needed available resources. Since whenever an available resource is needed that same resource is made available by the configurator it relates to, a subtyping relation can never be established. If we have type $\tau$ and another type $\tau'$ defined by the same configurator as $\tau$ composed with a method block $m$ whose type, after composition, merges the following to $\tau$:

$$\{r \bullet \{int\ get()\}\} \implies \{r \bullet \{int\ get()\}\}$$

for type $\tau$ to be a subtype of type $\tau'$ we would get the following derivation:

$$\frac{K \Longrightarrow K' \ <: \ L \Longrightarrow L', r \bullet \{int\ get()\}}{K \Longrightarrow K' \ <: \ L, r \bullet \{int\ get()\} \Longrightarrow L', r \bullet \{int\ get()\}}$$

that leads to an impossible relation since no rule exists to continue the derivation. This means that despite its possibility in theory, within ComponentJ it is not possible to come to a configurator whose type needs an available resource but doesn't introduce it. In this case, no subtyping relations can ever be established.

## 6.5 Dealing with name clashes

So far so good, and this subtyping relation seems to hold. However, what happens if, for instance, when replacing a configurator typed with the supertype by a one typed with the subtype, name clashes occur in the composition with another configurator? Should this situation be ignored? Should the "new" resources overcome the "old" ones? Or should the subtyping relation not hold?

Since the problem is when the subtype introduces more information than the supertype, the only rules that might be affected by this problem are 6.1 and 6.7. The other rules that establish a subtyping relation based on different sets of resources are not affected, since it is the supertype that has a larger set of resources, thus, when replacing the configurators, no new names can exist, therefore no name clashes occur.

Lets take rule 6.1, regarding the possibility of having new provided ports in the subtype. We know that provided resources are only added to the type by the *provides* configurator, and we also know that such a configurator is type not only with a *provided* resource but also with an *open* one. In the previous section it was established that no subtyping relation can exist between configurators if the *open* resources sets are not equal. This means that, to establish a subtyping relation between two types, where the subtype introduces more provided resources than the supertype, the subtype also

needs to include a way to fulfill the *open* resource (otherwise no subtyping relation can be established). The only configurator in ComponentJ that is able to fulfill open resources is *plug*. So, taking the following configurators $c_1$ and $c_2$:

```
provides I p;
```

*Configurator* $c_1$

```
provides I p;
provides I q;
plug x into q;
```

*Configurator* $c_2$

typed with types $\tau_1$ and $\tau_2$ accordingly:

$$\tau_1 : \{\} \Longrightarrow \{p \circ I, p \triangleright I\}$$

$$\tau_2 : \{x \bullet J\} \Longrightarrow \{x \bullet J, p \circ I, p \triangleright I, q \triangleright J\}$$

In this case rule 6.1 can be applied to establish the subtyping relation $\tau_2 <: \tau_1$. However, type $\tau_2$ also introduces and needs more resources than type $\tau_1$. Since there is no rule that allows the subtype to have more needed available resources than its supertype, then types $\tau_1$ and $\tau_2$ do not define a subtyping relation. However, if we introduce a method block definition with no external dependencies into configurator $c_2$:

```
provides I p;
provides I q;
methods x {...};
plug x into q;
```

type $\tau_2$ becomes:

$$\tau_2 : \{\} \Longrightarrow \{x \bullet J, p \circ I, p \triangleright I, q \triangleright J\}$$

In this case the relation between provided resources still maintains, and, by rule 6.7 we can establish the subtyping relation $\tau_2 <: \tau_1$.

Now take the situation where configurator $c_1$ is used to build n the following composition:

```
provides I p;
requires K x;
```

According to the suptype relation defined above, configurator *provides I p* can be replaced by the composition defined by $c_2$, since subtypes can be used instead of supertypes. By performing the substitution we get the composition:

```
provides I p;
provides I q;
methods x {...};
plug x into q;
requires K x;
```

As it can be seen, both method block and required port have the same name. At a first glance it might seem that this situation does not introduce any problems, since configurators *requires* and *methods* are typed differently and are used in different contexts (the plug operation know that it needs an available method block, and not a required port). However, as explained in chapter 4 the type of *x* in the *plug* operation is not defined. Name *x* will only be assigned a type when the type inference mechanism can figure out what type that is. If two resources have the same name, but refer to two different configurators, then the type inference mechanism becomes unsafe, since it might assign type *K* to name *x*, instead of the wanted type *I*. A possible way to deal with this situation is to take into account the order of the configurators in composition operations, and names are overridden whenever they are declared by another configurator. However, forcing an order in the composition operation can become cumbersome for the programmer, since some situations might not be easily detected.

This leads to the conclusion that, in what concerns the subtyping rule 6.1, the rule does not hold unless significant changes are applied to the language itself.

In what concerns rule 6.7, where a subtyping relation can be established between two types if the subtype introduces more *available* resources than its supertype, a similar process occurs. Take configurators $c_1$ and $c_2$:

```
requires I r;
```

*Configurator* $c_1$

```
requires I r;
methods p {...};
```

*Configurator* $c_2$

Assuming that method block $p$ does not depend on any external resources, the types of both configurators are the following:

$$\tau_1 : \{\} \Longrightarrow \{p \bullet I, p \triangleleft I\}$$

$$\tau_2 : \{\} \Longrightarrow \{r \bullet I, p \bullet I, p \triangleleft I\}$$

then, according to rules 6.7, the types of configurator $c_1$ and $c_2$ relate in a subtyping relation $c_2 <: c_1$. Now lets take a composition where configurator $c_1$ is used:

```
provides I p;
requires I r;
methods m {...};
plug m into p;
```

Because of the subtyping relation established above, it shouldn't be a problem to replace configurator $c_1$ by configurator $c_2$, in which case we get:

```
provides I p;
requires I r;
methods p {...};
methods m {...};
plug m into p;
```

As it can be seen, both provided port and method block are named $p$. In this case, the same problems arise as in rule 6.1: what type will be assigned to $p$ in the plug operation when the type inference mechanism runs? Can this problem be overcome? The answer is the same as before: it is not possible to overcome this problem unless significant changes are applied to the language.

With this discussion we can now reformulate the subtyping relation between configurator, being that it is now composed by the same subtyping rules as before, except rules 6.1 and 6.7.

# Chapter 7
# Validation of the Type System

***Validate:*** *to give official sanction, confirmation or approval.*

Testing compilers is not an easy task. Many approaches to validate implementations exist [21], however, for tests intended for the verification of implementations of the static and dynamic language semantics in a compiler, no algorithms for determining the reachability of the tests have been suggested.

To validate the type system implemented in this work we follow the approach in [6], where the testing process is split into three phases: correctness tests, real-world code and benchmarks. Correctness tests are simple tests to check if the compiler behaves as expected in very specific situations. If theses tests succeed, then the real-world code phase comes in, where larger system are implemented in a case-study kind of way. The last testing phase is used to measure the performance of generated code.

In this work only the first phase of tests is provided. A real-world example was outside the scope of this work, and is left for future work. This chapter provides the simple examples used to test ComponentJ's type system. A wider set of examples than the one presented in this chapter can be found with the compiler. To ensure the type system is working correctly every ComponentJ specific operation has been tested individually. This chapter is split into sections, each of them featuring the examples for some specific operation.

## 7.1 Type definitions

Type definitions are the basic operation in ComponentJ. Without them it wold not be possible, for instance, to define components provided or required ports.

### 7.1.1 Port interface

In what concerns port interface definition the only error detected is when the type has already been defined. In every other case no problems arise. An example of a correct port interface definition is:

```
port interface I {
    void set(int i);
    int get();
}
```

Since everytime a port is declared (either in a configurator or in any other type definition), from now on we will consider that port interface *I* exists with this same definition.

### 7.1.2 Object interface

Object interface type definitions include the provided port of the object. The proper way to define an object interface type is:

```
object interface OI {
    provides I p;
}
```

Three errors can be detected by the type system when dealing with this operation: a type with the same name already exists, the type of a provided port has not been defined, or there are at least two provided ports with the same identifier in the definition.

### 7.1.3 Component interface

Much like object interface definitions, typing errors in component interface definitions also have to do with undefined types of provided or required ports, of ports with the same name that already exist in the declaration, or of a type with the same identifier that already exists. The correct definition of a component interface can be:

```
component interface CI {
    provides I p;
    requires I r;
}
```

### 7.1.4 Script interface

In the case of script interface declarations, and although the type structure is different than any of the previous, the typing errors that might occur are the same as in object or component interface type definitions. The correct definition of a script interface type can be, for instance:

```
script interface SI introduces {
    available I m;
} needs {
    available I m;
    open I p;
}
```

From the next sections on more type verifications are made by the type system. Examples for each mistake detected by the type checker are provided, along with well-typed examples.

## 7.2 Component definition

The correct definition of a component means that the type resulting of the composition of the configurators used to define it does not have any needed resource nor does it introduce open resources. One possible component creation can be:

```
component C {
    provides I p;
    requires I r;
    methods m {
        void set(int i) { r.set(i); }
        int get() { return r.get(); }
    };
    plug m into p; }
```

The resulting type if the composition is:

```
script interface introduces {
    provided I p;
    required I r;
    available {void set(int), int get()} m;
} needs { }
```

meaning that the component *C* can be created, and will be type with the component interface type:

```
component interface {
    provides I p;
    requires I r;
}
```

Most of the errors in this section are not directly related to component creation, but to some configurator or composition between configurators. However, it is only when a component is created that these errors are reported, since until then not all information is available (configurators can be defined and only composed later on).

### 7.2.1 Unimplemented provided ports

One of the typing errors detected is when a component is created with unimplemented provided ports, i.e. when no implementation has been plugged to a provided port:

```
component C {
    provides I p;
    requires I r;
    methods m {
        void set(int i) {
            r.set(i);
        }
        int get() {
            return r.get();
        }
    };
}
```

This error is detected since the type resulting of the configurator composition still features one introduced open resource thus not passing the test.

### 7.2.2 Incompatibility between provided port and its implementation

The plug operation is intended to connect method implementations to the port where they are available. If for some reason one of the methods in the provided port is not provided with an implementation, then this error occurs. This means that the type of the provided port and the one of the method block connected to it are not compatible. One example of this case is:

```
component C {
    provides I p;
    requires I r;
    methods m {
        void set(int i) {
            r.set(i);
        }
    };
    plug m into p;
}
```

### 7.2.3 Undefined method block/provided port in plug operation

This type of error is detected in two situations: the first is when the method block used in the plug operation is not defined:

```
component C {
    provides I p;
    requires I r;
    plug m into p;
}
```

and the second when it is the provided port that is not defined:

```
component C {
    requires I r;
    methods m {
        void set(int i) {
            r.set(i);
        }
        int get() {
            return r.get();
        }
    };
    plug m into p;
}
```

Both these cases are detected since the type of composition of all configurators has needed resources (available in the first case and open in the second). Since for a component to be created no needed resources can exist, an error is thrown.

### 7.2.4 Required port not available

Whenever a port is used in the definition of a method block inside a component, then that same port must be available at the component creation moment. In this case, that means that a required port must be introduced in the component, so that it becomes available. If, for instance, we have:

```
component C {
    provides I p;
    methods m {
        void set(int i) {
            r.set(i);
        }
        int get() {
            return r.get();
        }
    };
    plug m into p;
}
```

then the resulting type of the composition between configurators will have needed resources since the type of the method block features one needed available resource and there is no introduced resource to match it in any of the other configurators.

For the next set of examples lets take a configurator *C1* defined as:

```
component C1 {
    provides I p;
    methods m {
        int v = 0;
        void set(int i) {
            v = i;
        }
        int get() {
            return v;
        }
    };
    plug m into p;
}
```

## 7.3 Object instantiation

When instantiating an object from a component all required ports of the new object must be connected to some ports provided by other objects, ensuring the well functioning of the newly instantiated object. Following this rule, and taking the previous components, we get:

```
OI o = new C1;
OI oo = new C with[r := o.p];
```

as being a valid instantiation, since component *C1* does not have any required port, and its provided port is connected to the port *r* required by *C*.

### 7.3.1 Variable type undefined or incompatible

To be able to use an instantiated object a new variable must be created, whose type is an object interface type that must previously be defined. If the type assigned to the variable is undefined then a typing error occurs:

```
OOI o = new C1;
```

In this case type *OOI* has not been defined, so variable *o* cannot be assigned a type, leading to a typing mistake.

If, on the other hand the variable's type does exist, then a compatibility test must be executed. If the variable type is compatible with the type of the instantiated object (they are either the same or in a subtyping relation where the object type is subtype of the variable type) then the assignment is made. If, however, they are not compatible then a typing error is thrown. One example of this case is when doing the following:

```
object interface OI1 {
    provides I p;
    provides I q;
}

OI1 o = new C1;
```

since the type obtained from the instantiation is:

```
object interface OI1 {
    provides I p;
}
```

and it not a subtype of OI1.

### 7.3.2 Component undefined

Only existing components can be instantiated, so, if one tries to instantiate an unexisting component, such as in:

```
OI o = new C2;
```

an error will occur, since *C2* does not exist and therefore cannot be instantiated.

### 7.3.3 Required ports incompatible with assignments

If a component has some required ports, at the time of its instantiation the required ports must be connected to provided ports of other objects. This is done by means

of the with clause. In this case three scenarios might occur: all required ports are connected, at least one required port is left unconnected, or, at least, more connections then the ones required are made. The first case represents a correct instantiation like the one presented before. The second and third cases represent mistyped instantiations. Examples of both these errors can be:

```
OI oo = new C;
```

where the required port *r* of component *C* is left unconnected, and:

```
OI o = new C1;
OI oo = new C with [r := o.p, q := o.p];
```

where one inexistent required port *q* is being connected to provided port *p* of object *o*.

## 7.4 Component definition with inner components

The use of inner components is possible in ComponentJ by means of the *uses* configurator when creating a component. Take a new component interface *CI1*:

```
component interface CI1 {
    provides I p;
}
```

To create a new component *comp* whose type is the one of *CI1* then the following is done:

```
CI comp = compose(
              provides I p;
              uses r = C1;
              methods m {
                  void set(int i) { r.p.set(i); }
                  int get() { return r.p.get(); }
              }
              plug m into p;
          );
```

This means that component *C1* (defined before) is an inner component of *comp*, thus its provided ports are available inside *comp* through the name *r*. So, in the method block definition, when using r.p it means the port *p* of component *C1*, identified by *r*. The same typing mistakes detected by the type system in this operation are the same as in component definitions without inner components.

## 7.5 Dynamic reconfiguration

The type check of the dynamic reconfiguration operation is performed at runtime, and is based on the runtime type information available for both the object being reconfigured and the configurator used to reconfigure it. Because it is a runtime test if a reconfiguration cannot be applied to an object, then the system must remain the same, and the operations in the *else* branch are executed.

As an example let's take a simple reconfiguration operation, based on the previous examples in this chapter, where it is intended to add a new provided port *np* of type *I* to object *o*. The configurator to perform this action could be (assume *T* to be the type of the configurator):

```
T conf = (
    provides I np;
    methods mb {
        void set(int i) {...}
        int get() {...}
    }
    plug mb into np;
);
```

Using this configurator to reconfigure object *o* is done as follows:

```
reconfig o using conf as o2 in {
...
} else {
...
}
```

This operation means that the runtime system will try to apply configurator *conf* to

object *o*. If it succeeds then the operations in the *in* branch are performed, and *o* will be identified as *o2* inside that branch. In this case, and because it is a simple example where the program flow can be predicted, no problems arise from the application of the configurator to the object. If, however, instead of configurator *conf* we had, for instance, configurator *conf2*:

```
T conf = (
    provides I p;
    methods mb {
        void set(int i) {...}
        int get() {...}
    }
    plug mb into p;
);
```

there would be a provided port overlap leading to a mistake (it is not possible to have two provided ports identified by the same name). In this case the object would remain the same and branch *else* would be executed.

# Chapter 8
# Concluding Remarks and Future Work

ComponentJ is a component-based programming language that seeks to provide a simple way to perform component creation and composition, as well as dynamic reconfiguration of component structures. To ensure the type safety of all operations, ComponentJ also features a type system that allows the compile-time detection of possible defects in the components/objects structures.

In what concerns component related operations, the type checker ensures the good structuring of created components and objects, as well as object networks. This guarantees that, when the system is executing, no problems will arise due to deficient definitions, leading to, for instance, method-not-found errors. Also, and because of ComponentJ's strong notion of information hiding, a dynamic type verifier must be used to ensure the type safety of the reconfiguration operation. This means that, after a reconfiguration, the runtime structure of the object remains stable.

The main goal of this dissertation was to provide an implementation of such a type system, and integrate it in the existent compiler prototype [35]. To improve the programming process by allowing some type information to be omitted from the source code, a type inference mechanism, based on the Damas-Milner algorithm for type inference [17, 24] was introduced in the implementation. This mechanism is mainly used for plug operations and method block definitions, since configurators can be defined outside the scope of a component, and external resources might not be available when the configurator is being defined. The type inference mechanism delays the verification of the external resource types by assigning each structure a type variable that will, later on, be instantiated into a specific type.

Apart from the implementation of the type system, a subtyping relation between configurator types, not studied in the original calculus, has also been discussed, however no formal proof of its correctness is provided. Such a relation increases the reliability of systems, as it provides a way of increasing functionality without having to change the original system. This discussion of such a subtyping relation states that, for

a subtyping relation between two script interface is very limited due to possible name clashes, and only exists between configurators whose types only differ in introduced required resources and needed available resources.

As a way of validating the implementation a set of examples testing possible outcomes, correct or faulty, is used. These examples test very specific operations of the programming language (only ComponentJ specific operations), ensuring the outcome is as excepted.

The first main challenge that had to be overcome in this work was to understand the compiler prototype implementation where the type system would be included. Understanding the different statements and expressions, and the code each expression generated became a crucial step to the correct implementation of the type system. Besides that first step in this work, finding the suitable data structures to represent the different types within ComponentJ was also a big challenge, mainly in what concerned script interface types, since it is the type with the more complex structure.

As a result of this work, as well as of the previous work made in [28, 35] resulted a paper presented in a national conference [33] and an extended version of the same in an international journal [34].

## 8.1 Future Work

ComponentJ is still a work in progress, so many aspects can be improved and new features can be introduced for the language to become more and more attractive to new users.

The validation of the implementation has only been made by means of simple very specific examples, as presented in chapter 7. To ensure the well-functioning of the ComponentJ's compiler and type system a larger set of examples must be made, including some real-life, more extensive examples, that not only provide a means of validation, but also show how ComponentJ can be used in larger software systems.

Related to the type system itself, the subtyping relation introduced in this dissertation does not include formal proof of its correctness, nor is it present in the implementation. Having this relation verified and implemented would increase even more the flexibility of the language.

One aspect that can be improved is the creation of an extensive library with ready-to-use components. Component-based software is based on the principle of software reuse, so, having an online component repository, and easy ways to access it would allow programmers to easily reuse components created by others.

Since ComponentJ provides a way to perform runtime reconfigurations of a system, it would be interesting to make use of this feature to allow automatic dynamic software updates. This could be a process with very little human intervention (if any), since the type checker already prevents faulty reconfigurations.

# Bibliography


[1] Abstract Window Toolkit, May 2008. http://java.sun.com/products/jdk/awt/.

[2] Enterprise JavaBeans, May 2008. http://java.sun.com/products/ejb/.

[3] Java Swing, May 2008. http://www.javaswing.net/.

[4] JavaBeans, May 2008. http://java.sun.com/javase/technologies/desktop/javabeans/.

[5] Standard Widget Toolkit, May 2008. http://www.eclipse.org/swt/.

[6] Testing the c++ compiler, February 2009. http://blogs.msdn.com/fivetestersfromvc/archive/2004/02/10/70438.aspx.

[7] Jonathan Aldrich, Craig Chambers, and David Notkin. Architectural Reasoning in ArchJava. In *ECOOP '02: Proceedings of the 16th European Conference on Object-Oriented Programming*, pages 334–367, London, UK, 2002. Springer-Verlag.

[8] Jonathan Aldrich, Craig Chambers, and David Notkin. ArchJava: Connecting Software Architecture to Implementation. In *Proceedings of the 24th International Conference on Software Engineering*, pages 187–197, New York, NY, USA, 2002. ACM.

[9] D. Ancona, G. Lagorio, and E. Zucca. Smart modules for Java-like languages. Technical report, Universita di Genova, 2004.

[10] Davide Ancona and Elena Zucca. True Modules for Java-like Languages. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 354–380, London, UK, 2001. Springer-Verlag.

[11] E. Bruneton, T. Coupaye, M. Leclercq, V. Quema, 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.

[12] E. Bruneton, T. Coupaye, M. Leclercq, V. Quema, 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.

[13] Luca Cardelli. Typeful Programming. In E. J. Neuhold and M. Paul, editors, *Formal Description of Programming Concepts*, pages 431–507. pringer-Verlag, Berlin, 1991.

[14] Luca Cardelli. *Type Systems*, chapter 103. CRC Press, Boca Raton, FL, 1997.

[15] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 17(4):471–523, 1985.

[16] D. Chappel. Introducing SCA. Technical report, Chappell & Associates, 2007.

[17] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212, New York, NY, USA, 1982. ACM.

[18] Dominic Duggan and Constantinos Sourelis. Mixin modules. In *ICFP '96: Proceedings of the first ACM SIGPLAN international conference on Functional programming*, pages 262–273, New York, NY, USA, 1996. ACM.

[19] Miguel Durão. ComponentGlue: Uma Linguagem de Composição para Arquitecturas Distribuídas. Master's thesis, Universidade Nova de Lisboa, Faculdade de Ciências e Tecnologia, 2004.

[20] 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.

[21] A. S. Kossatchev and M. A. Posypkin. Survey of Compiler Testing Methods. *Programming and Computer Software*, 31(1), January 2005.

[22] 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.

[23] 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.

[24] Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[25] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, Cambridge, Massachusetts, 2002.

[26] František Plášil, Dušan Bálek, and Radovan Janeček. SOFA/DCUP: Architecture for Component Trading and Dynamic Updating. pages 43–52. IEEE CS Press, 1998.

[27] J. A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of ACM*, 12(1):23–41, 1965.

[28] João Costa Seco. *Language and Types for Component-Based Programming*. PhD thesis, Universidade Nova de Lisboa, Faculdade de Ciências e Tecnologia, 2006.

[29] João Costa Seco and Luis Caires. A Basic Model of Typed Components. Technical report, Departamento de Informática - Universidade Nova de Lisboa, 2000.

[30] João Costa Seco and Luís Caires. A Basic Model of Typed Components. In *ECOOP '00: Proceedings of the 14th European Conference on Object-Oriented Programming*, pages 108–128, London, UK, 2000. Springer-Verlag.

[31] João Costa Seco and Luís Caires. Componentj: The reference manual. Technical report, UNL-DI, Monte da Caparica, Portugal, 06 2002.

[32] João Costa Seco and Luís Caires. Types for Dynamic Reconfiguration. In *ESOP '06: Proceedings of the European Symposium on Programming*, pages 214–229, London, UK, 2006. Springer-Verlag.

[33] João Costa Seco, Ricardo Silva, and Margarida Piriquito. ComponentJ: a component-based programming language with dynamic reconfiguration. In *Proceedings of Compilers, Related Technologies and Applications*. Polytechnic Institute of Brangança, Portugal, 2008.

[34] João Costa Seco, Ricardo Silva, and Margarida Piriquito. ComponentJ: a component-based programming language with dynamic reconfiguration. *Computer Science Information Systems*, 5(2), December 2008.

[35] Ricardo Silva. Compilador e Sistemas de Suporte à Execução da Linguagem de Programação ComponentJ. Technical report, Universidade Nova de Lisboa, Faculdade de Ciências e Tecnologia, 2008.