

# Type Safe Composition in .NET

João Costa Seco\*Departamento de Informática  
Universidade Nova de Lisboa  
Joao.Seco@di.fct.unl.pt

## Abstract

We discuss the design and implementation of a composition language for the .NET platform. The language is based on a simple core language that includes specific abstractions for composition. We explain some of the design options taken and discuss the representation of structured components, dynamic composition, type safe loading and sharing of types between components.

## 1 Introduction

In this paper, we describe the application of a programming language based approach to safe component composition to the .NET platform. Our proposal builds on a core calculus we have been designing to express some fundamental features of component-oriented programming [20, 21]. Our concern has been the study of the integration of the composition constructs available in our core calculus into a practical programming language, while preserving the safety properties enforced by the former. Our aim now is to discuss some design issues taken in the implementation of **Component C<sup>#</sup>** (CC<sup>#</sup>), a glue language for safe component composition for the .NET environment. We focus the discussion on the mapping of the component model to the CLR’s object model and on the use of component oriented aspects of the CLR like custom attributes.

So far, most of component based programming has been centered in “standards” like COM, CORBA or JavaBeans. These models provide fundamental features like dynamic loading of modularization units, and specification through interfaces or specific description languages (IDLs). With the appearance of .NET, many of these features are directly implemented at the level of the execution layer. This fact may by itself motivate the development of component-oriented programming languages. Additionally, we can also find new interesting features like a general purpose intermediate runtime language, a common type system, and the organization of the code into assemblies tagged with manifest information. Components written in different programming languages can then

---

\*Work partially supported by a national project, “DataBricks” Sapiens Project #33924, 2000.

be compiled to the same virtual machine, and interact without complex marshaling processes. As in many other component models, component programming in .NET is mostly regarded as a software engineering practice, where composition is usually implemented by the ad-hoc aggregation of objects, instead of being the result of instantiating separately specified architectures. Moreover, although important issues like external dependencies and version control have already been addressed to some extent [14], we believe that another step toward true component orientation is still missing, namely, the support to composition by suitable programming abstractions.

There are a variety of aspects arising in component integration, for instance behavior and performance constraints, our current focus is on type safe composition. However, we adopt a common integration scheme that might serve as a basis to support in the future other kinds of requirements. The availability of a specification for the border-line of a black-box component is crucial to evaluate its suitability to a given context. Additionally, having explicit specifications of architectures and components is useful not just for documentation purposes but also to support static analysis procedures, able to determine e.g., whether all architectural elements are properly integrated or if the services each element provides and requires are used in a consistent way.

To integrate such requirements at the programming language level we designed a simple core language [20, 21] that include specific abstractions for composition. In our model, components are first-class entities, and the basic constructs of the calculus enable composition, adaptation of existing components, and the definition of basic functionality in an object-oriented style. The calculus is equipped with a type system that assigns types to both *components* and *objects* (component instances) and validates composition constructs. Then, the type system is able to check architectural properties of component structures that would be difficult to check in more standard approaches, where the composition code (composing objects, not components) is mixed with the user code. Parametric polymorphism at the component level and structural subtyping are other features of the type system aimed at improving reuse. In this paper, we discuss some issues that arise during the porting of our model to the .NET platform.

## 2 Composition in programming languages

In the history of programming, structuring mechanisms have frequently made their first appearance as programming idioms of existing programming languages. Then, mechanisms that reveal themselves as simple, usable, and really fundamental, tend to generate new programming language constructs where they appear as primitive concepts [3, 11]. Our aim is to develop programming language primitives able to express composition, and to allow the safe manipulation of components at run-time as first-class entities [19].

Although we may argue that the concept of software component is not far from the traditional notion of module, and that a quite extensive amount of

work has been developed on modularity in programming languages (mainly in functional languages) [4, 5, 7, 15, 16], the need to address specific requirements of component-oriented programming seems to have appeared only recently. Even powerful module languages such as SML, recently ported to .NET [9], do not seem to provide a smooth correspondence between SML modules, SML module composition, and .NET components.

Component-oriented programming, at least object-based component programming, is usually based on the definition of object aggregates relying on cross-references to implement the intended architectures. Definition of a component's architecture in frameworks like JavaBeans, COM, or even in .NET is still based on the definition of object aggregates. These mechanisms are unsafe in the sense that the implementation of a component specification must rely solely on the careful programmer to ensure the correctness of certain assignments or constructor calls, and not on a mechanical verification performed by some compiler. Properties like the effective implementation of a service or the satisfaction of internal dependencies must then be ensured manually. Although the assembly mechanism present in .NET already identifies external dependencies of components, the existence of needed components is not performed by the run-time until they are actually used. This may cause an execution failure that could be prevented in the loading process.

Composition is considered as a powerful way of decoupling software systems while avoiding anomalies caused by mechanisms like implementation inheritance [23]. A programming language that uses black-box composition as the preferred extension mechanism avoids problems such as the fragile base class problem [6] and brings to the programming language level guidelines that Component Base Software Engineering recommend. Since  $CC^\sharp$  is intended to design component architectures, it is used at a stage where inheritance is no longer usable as a reuse mechanism, where the "is-a" relationship is artificial and the aggregation of instances becomes a common pattern. Composition and Software Architectures have been studied formally as a software engineering practice using tools that verify and implement the architectures [12]. However, there is a large distance between the component model and the implementation of its components and their correspondence is not easily verified.

Recently, other authors also proposed the integration of composition primitives in programming languages. Ibrahim developed a formal model with a type system for COM exemplified in the model language COMEL [18]. It represents aggregations and containment of components and gives a type to each component by joining a set of interfaces. However, it does not have the notion of component dependencies. ArchJava [1] and ACOEL [22] also bring composition to a programming language level. These languages support some composition constructs and allow for the verification of architectures, however they fail to enforce a clear separation between adaptation and composition code in programs. Zenger's component calculus [24], built as an extension of *Featherweight Java* [8], focus on evolution and extension of components rather than on architecture definition. This proposal is an interesting attempt to conciliate inheritance with composition; it also introduces a type system close to ours, but without support

for type parametrization.

Several approaches to the introduction of modules in programming languages are also related to component programming. For instance, Jiazzi [13] is an adaptation of the *MZScheme Units* model for building modules of Java classes. Each module is defined statically as an atomic element or a compound, importing and exporting some java classes according to a specification. Jiazzi compounds are not used as first-class values and therefore dynamic composition is not yet considered in this model. JAVAMOD [2] adds a compositional layer to Java in a way close to module languages like ML. It keeps the module composition language separate from the operational code that provides the basic functionality of an application, and treats modules just as compile time values. Our component calculus mixes both operational and compositional aspects in the very same language while still allowing such aspects to be expressed in separate syntactic contexts. We think that this provides a better support for the adaptation of component functionality.

### 3 The component calculus

In this section we describe the formal component calculus introduced in [20, 21]. It is a typed imperative calculus with primitives for constructing polymorphic components. The purpose of the calculus is to study the properties of composition in programming languages and to support the development of component-oriented scripting and programming languages. The syntax of a calculus is shown<sup>1</sup> in Fig. 1. It includes basic constructs of an imperative object oriented calculus. Furthermore, specific constructs are introduced to support the manipulation of components and of their instances. This is the case of the **compose** expression which denotes a component value with an architecture specified by the enclosed expression. Component values can then be used in instantiation expressions (**new** – **with** – ) to produce objects which are the run-time entities that implement the specified functionality. Following the types of the language an object is typed as a record of *ports*. A port is typed by an interface, and thus corresponds to an access point to a set of methods. An instantiation expression also needs to define where to find the services that a component needs to be instantiated, references to such services must be provided by the bindings that follow the keyword **with**. Thus, our core language specifies both computation and architecture while maintaining them separate in the code, and therefore improving its readability.

In the definition of a new component by a **compose** expression, a list of type parameters defines bounded type variables in the scope of the component expression, and component expressions define the different aspects of the component's architecture. In general, each component expression denotes a component with a specific feature that is further specified by the inner expression.

The expression (**requires**  $x : I$  and  $c$ ) adds to the component denoted by  $c$  a required service port  $x$  with interface type  $I$ . This means that the resulting

---

<sup>1</sup>We use the notation  $\Phi_i$   $i \in 1..n$  to denote a sequence of expressions  $\Phi_1, \dots, \Phi_n$ .

$\tau$	$::= I \mid P \mid C \mid \text{Var}(\tau)$	(Types)
$I$	$::= \{m_i : \gamma_i^{i \in 1..n}\}$	(Interface)
	$X$	(Type Variable)
	$\mu X.I$	(Recursive Type)
	$\text{Top}$	(Common Supertype)
$R, P$	$::= \{p_i : I_i^{i \in 1..n}\}$	(Port Type)
$C$	$::= \langle X_i \leq I_i^{i \in 1..n} \rangle R \Rightarrow P$	(Component Type)
$\gamma$	$::= (\tau_i^{i \in 1..n}) \tau$	(Method Type)
$e$	$::=$	(Expressions)
	$x$	(Variables)
	$e.m(e_i^{i \in 1..n})$	(Method call)
	$!x$	(Ref Value)
	$x := e$	(Ref Assign)
	$\text{let } x = e \text{ in } e$	(Local Name)
	$\text{compose } \langle X_i \leq I_i^{i \in 1..n} \rangle c$	(Composition)
	$\text{new } e \langle I_i^{i \in 1..n} \rangle \text{ with } p_j := e_j^{j \in 1..m}$	(Component Instantiation)
	$e.p$	(Port selection)
$c$	$::=$	(Component Expressions)
	$\varepsilon$	(Empty Component)
	$\text{requires } x : I \text{ and } c$	(Required Port)
	$\text{provides } x : I \text{ and } c$	(Provided Port)
	$\text{uses } x : C = e \langle I_i^{i \in 1..n} \rangle \text{ and } c$	(Inner Component)
	$\text{declare } \{a_i : \tau_i^{i \in 1..n}\} \text{ and } c$	(State Variables)
	$\text{methods } x =$	
	$\quad \{m_j(x : \tau) : \tau_j = e_j^{j \in 1..m}\} \text{ and } c$	(Method Block)
	$\text{plug } \pi \text{ into } \pi \text{ and } c$	(Port Connection)
$\pi$	$::= p \mid x.p$	(Plug Expression)

Figure 1: The calculus: types and abstract syntax.

component will require access to a suitable service provider in order to be instantiated. The required port name  $x$  is then available inside  $c$  for direct method calls and to connections with compatible destinations. On the other hand, the expression (**provides**  $x : I$  **and**  $c$ ) denotes a component whose instances provide a service at port  $x$  of type  $I$ . These two kinds of expressions define the component boundary by adding required or provided ports, together with their types, to the type of the resulting component.

The remaining forms of component expressions allow the introduction of inner elements into the architecture being defined. The expression (**uses**  $x = e : C$  **and**  $c$ ) introduces an inner component which is referred inside  $c$  by the local name  $x$ . Such inner component is specified by the expression  $e$ , that must evaluate to a component value of a subtype of  $C$ . Primitive behavior and scripting of predefined components is obtained by the method block expression (**methods**  $x = \{ \dots \}$  **and**  $c$ ). Although components are values, the state variables declared in the expression (**declare**  $\{a_i : \tau_i^{i \in 1..n}\}$  **and**  $c$ ) will belong just to their instances. Once the elements of an architecture are introduced, they must be interconnected between themselves and to the declared provided and required services. The connection of ports is realized by **plug** expressions (**plug**  $\pi_1$  **into**  $\pi_2$  **and**  $c$ ). The source ( $\pi_1$ ) must be an available service: a required port of the component being defined, a provided port of some inner component, or a method block of the component. The destinations ( $\pi_2$ ) is some service that must be fulfilled inside the component to achieve component consistency: for instance the required ports of an inner component, or a provided ports of the component being defined. Available services can also be used transparently in the method blocks of the component.

The visibility rules implemented by the type system define that all available services in the component can be used in the component and are not visible outside its borders. They also state that the only names that can cross a component's boundary are the ones denoting component values. This allows the use of component values in new compositions and places all the dependencies at the component level.

### 3.1 Type safe dynamic composition

To illustrate the use of our core programming language, we pick an example from [17] of component programming in a dynamic application scenario. This example allows us to show how the features of the calculus are used in dynamic application environments, while enforcing certain consistency properties of the components.

The scenario is a surveillance system with a server that has access to a certain number of sensors (cameras and other detectors) and transmits messages to the security staff. They carry equipment like pagers, mobile phones, or PDAs that communicate through various means (IRDA, wireless LAN, GPRS/GSM, etc.), or they have access to desktop computers in a LAN. The number and type of messages that are sent may vary according to the connections and missions assigned to each security guard. Hence, the overall architecture of the system

consists of one server communicating with different kinds of clients, with different communication capabilities. Thus, the service provided to clients must adapt itself to the characteristics of the different client devices, while maintain the same functional specification. The modularization of this system demands that different implementations are used to attend each situation instead of a complex implementation capable of producing all the information encodings.

In the solution discussed in [17], a special entity called “configurator” reads the description of a configuration in a XML-based language, checks the environment through services available in the system, and then assembles a new object able to deal with the scenario at hand by interconnecting several instances of existing components.

Our proposal is to produce pre-configured components that can be instantiated as a whole instead of aggregating instances obtained independently. We represent the architecture of the service handler on the server side by an “incomplete” composition whose missing elements are fully specified and can be chosen based on profile and environment conditions; once completed, the particular architectures can be instantiated safely.

Consider the case where the surveillance service has an architecture which varies on two components according to the situation: a media composer, used to translate server events to proper media (pager code, SMS text, still images, video clips, video stream), and a message filter, used to decide which events a user should receive. In Fig. 2 we show the specification of a component (*configurator*) that exposes a factory method (*makeServer*) returning a component value with the described architecture using the method parameters to complete it. The different roles (a filter and a media composer) must be fulfilled by existing components, and passed as arguments to the method. The values returned by this method are components configured to handle the specific situation at hand. Another method (*selectServer*) makes the assessment of the current environmental situation, based on its port required (*sensors*), and then decides what components to use.

In the code fragment shown, the *configurator* component declares that its instances require a service of type *ISensors* at port *sensors* and provide a service of type *IConfig* at port *config* (in Fig. 3) we show the type abbreviations used in the code. The services provided by port *config* of type *IConfig* are implemented in the method block *m* (last *plug* expression at the bottom of Fig.2). The implementation of the method exposed in port *config* (*selectServer*) relies on the method *makeServer* that builds the desired architecture on demand. The uses expressions that appear in the body of method *makeServer* introduce placeholders (*filter* and *media*) that will be bound to definite components passed as arguments to the factory method. Thus, method *makeServer* returns a newly created component for each appropriate combination of inner elements.

The instances of a server component are notified of occurring events by means of a method call to *notify*, available in their *observer* port as in the observer pattern. This behavior is specified in interface *IServer*, and implemented by the methods defined in the method block *s* inside the composition expression in method *makeServer*. These methods will make use of the yet to

```

let configurator = compose (
  provides config : IConfig and
  requires sensors : ISensors and
  methods m = {
    selectServer(user : IProfile, c : IConnection) =
      //decide which components to use based on
      //the bandwidth and the user category
      let someFilter = ...
        in let someMedia = ...
          in m.makeServer(someFilter, someMedia),
    makeServer(myFilter : CFilter, myMedia : CMedia) =
      compose (provides observer : IServer and
        requires sys : ISystem and
        uses filter = myFilter and
        uses media = myMedia and
        declare {con : IConnection} and
        methods s = {
          notify(e : IEvent) =
            ...con.send(media.compose.getMessage(e)),
          login(c : IConnection) = con := c...
        } and
        plug s into observer
      )
    } and
  plug m into config
) in ...

```

Figure 2: The *configurator* component

be determined *myFilter* and *myMedia* to implement its functionality, however its type soundness will be granted just by looking to statically available type information.

Finally, in Fig. 4, we show how the *configurator* component can be instantiated provides the component all of its requirements. We can here identify two situations. In the first one, a security guard holding a cell phone logs into the system and its mission is a predetermined route outside the building. Given the user identification and the equipment parameters, the method *selectServer* chooses an appropriate filter that restricts the messages to the ones related to the predetermined route, and a media composer that extracts from the server text messages and translates them to SMS messages, and then calls *makeServer* to compose a component with those elements. In the second one, an user logs in from a desktop PC in the central office and in this case it would be more appropriate a less restrictive filter and a media composer that sends messages

```

IEvent ≡ ...
IConnection ≡ ...
IProfile ≡ ...
IServer ≡ {notify : (e : IEvent) unit, login : (c : IConnection) unit}
CServer ≡ {s : ISystem} ⇒ {o : IServer}
ISensors ≡ ...
IConfig ≡ {selectServer : (user : IProfile, c : IConnection) CServer}
CFilter ≡ ...
CMedia ≡ ...

```

Figure 3: The type abbreviations used

containing (where available) video streams that pop up in the screen of the PC. The resulting components have the same component type and can be used in the same server context. This allows the reuse of the *configurator* in an evolving set of profiles, filters and media composers.

Our example illustrates the dynamic configuration of a server, but not the extensibility of the configurator with respect to the possible architectures to be produced. However, it would not be hard to factor the *selectServer* method, which chooses the inner elements, and change it to deal with different architectures, selecting them dynamically as the other elements. The collection of factory components could then be kept in a component repository.

The key advantage of our approach is that the architectures are type checked and the resulting server components are ensured to be consistent. According to the type system, compositions resulting from the method *makeServer* are consistent if the inner components are typed with subtypes of *CMedia* and *CFilter*, which is ensured by the typing of the method call expression and by the subtyping relation between component types. Thus, the flexibility of the original component-oriented design is completely preserved in our implementation, however the use of type safety at the architectural level contributes to improve the code quality and readability.

```

let c = new configurator with sensors := ... in
...
let s1 = c.observer.select(exteriorGuard, cellPhoneConnection) in
...
let s2 = c.observer.select(officeGuard, desktopPCCConnection) in

```

Figure 4: Using the configurator component

```

port interface IFloatUnaryOp {
    float compute(float r);
}

component Square {
    provides IFloatUnaryOp op;
    methods m {
        float compute(float r) {return r*r;}
    }
    plug m into op;
}

```

Figure 5:  $CC^\sharp$  source code

## 4 Porting the calculus to .NET

In this section, we discuss the design and implementation of *Component C<sup>‡</sup>*, a composition and scripting language for .NET based on our component calculus.  $CC^\sharp$  is a type safe glue language able to manipulate components as first-class values in computations and dynamic compositions, that includes an expressive scripting language (in this case a subset of  $C^\sharp$ ).

We follow an approach close to the one we have used in the implementation of *ComponentJ* for the Java environment. This first phase of the definition of  $CC^\sharp$  is limited to the port of the composition primitives using the mechanisms provided by the underlying platform. A next phase of the language design will focus on other language features and constructions, like component constructors, other composition patterns, and a wider integration with the class framework.

Our current implementation technique is by source translation: the  $CC^\sharp$  compiler type checks and translates  $CC^\sharp$  source files into  $C^\sharp$  source files that can then be compiled by the standard  $C^\sharp$  compiler. A  $CC^\sharp$  source file consists of a sequence of type declarations and of static component declarations.  $CC^\sharp$  components may also result from the evaluation of anonymous **compose** expressions defined inside method bodies. In Fig. 5 we show a  $CC^\sharp$  source file declaring a simple component called **Square** whose instances provide a service typed by the interface type **IFloatUnaryOp**, through port **op**.

### 4.1 Mapping compositions to $C^\sharp$

The compilation of  $CC^\sharp$  relies on a minimal runtime system consisting of framework of classes and interfaces providing some basic support for loading and identifying components, and on the definition of an interaction protocol for components and component instances. We first describe the translation of a component’s structure to  $C^\sharp$ , followed by the algorithm used in the initialization of a component instance. We then discuss aspects related to the interoperation of our component model with “pure” .NET components, the declaration and

```

class Square : CCSCOMPONENT {
    Square() {}
    static CCSCOMPONENT value = new Square();
    CCSCOMPONENT valueOf() { return value; }
    object createInstance(...) {...}
    class Instance : CCSInstance, ... {...}
}

```

Figure 6: The mapping of component `Square` to  $C^\sharp$  (Fig. 5)

implementation of dynamic compositions, and the implementation of generic (type parametric) components.

**Representing  $CC^\sharp$  components** To represent  $CC^\sharp$  components we use the Factory and Singleton design patterns. Each  $CC^\sharp$  component is encoded by (1) a  $C^\sharp$  class with a factory method that produces component instances, and (2) a singleton that yields the runtime value representing the component as a first class entity. We will refer to this class as the *component class*. Nested inside the component class we define another class implementing the component’s functionality and used to generate the component instances. We will refer to this inner class as the *instance class*.

In Fig. 6 we illustrate the mapping of component `Square` from Fig. 5 to the corresponding  $C^\sharp$  class. Class `Square` extends the class `CCSCOMPONENT`, from the  $CC^\sharp$  framework, and implements the factory method (`createInstance`) and the singleton value accessible through the `valueOf` method.

Inside the instance class (`Square.Instance`), the component’s architecture is instantiated into an aggregate of objects whose consistency was statically checked. Components are recursively instantiated creating a tree where each instance holds references to the objects that result from its inner elements: components or method blocks. They have exclusive references to the enclosed objects which are interconnected by a network of cross-references in the class constructor according to the component’s architecture. However, the interconnections between the objects of an architecture can form a complex network with loops and temporarily uninstantiated connections (due to disconnected required ports in the middle of the instantiation process). In this scenario a direct assignment strategy does not work. Therefore we build the network in a two stage algorithm: In the first phase, we use *port provider* objects, one for each plug source and destination, to establish the connections at a meta-level. Port provider objects represent delayed connections that form chains. The second phase of the algorithm simply follows all the chains and fills the ports with references to the method blocks that actually implement the expected functionality.

Port provider objects are the chain elements that can hold references to a source, continue the chain to another provider, or simply be empty. The type

```

component interface TFloatUnaryOp {
    provides IFloatUnaryOp op;
}

```

Figure 7: A component type expression

system of the calculus ensures that, at instantiation time, these chains are either complete, which means the end is a method block, or they have malicious cycles that result in endless recursions [20]. Each component instantiation makes its meta-level of connections based on the assumption that all the instances of inner components are already connected inside, and that the visible required and provided ports are part of valid chains. Instances representing method blocks are placed at the end of new chains by instantiating new port providers, and required ports are also assigned to new port providers with empty following (they are placeholders for future connections).

The implementation of this algorithm is supported by fields in the instance classes, representing each port provider. It connects all the providers in the constructor of the instance class according to the component’s architecture. The second phase of the algorithm is implemented in a method that follows the chains and initializes the fields that represent the ports (`initPorts`) with references that really implement the corresponding services.

**State variables** In components expressed in the component calculus, the implementation of the services provided by any component instance share a common state, and both method blocks and state variable blocks are represented by independent constructs (see Fig. 1). However, in  $CC^\sharp$  we combine the declaration of state (variables) and functionality (methods) in a single construct, thus limiting the scope of state variables to the enclosing block. Sharing of state variables between method blocks must then be done explicitly through method calls, which in turn may be automatically generated by the compiler. A construct similar to object constructors common in class-based object-oriented languages can then be used to pass values to newly created component instances, or to redirect values in an hierarchal manner through an architecture. In any case, such initialization code for component instances can already be easily compiled down to our core calculus.

**Component compatibility** The integration of .NET components in compositions written in  $CC^\sharp$  is made by using a *skeleton* class, implementing the programming pattern described here, and using it to wrap the components built on other programming languages. This skeleton class is produced from a component type expression (like `TFloatUnaryOp` in Fig. 7). It implements all the factory and singleton functionality and is extensible letting the internal functionality of the component to be defined by the programmer.

**Dynamic compositions** Dynamic compositions are unevaluated architectures with missing elements. The static type of the elements is known and therefore they may be compiled into a class but some inner components are unknown at that time. Each time the composition is evaluated the missing elements must be completed by storing references to component values. Instances of those components are then obtained from their factory methods. The translation of dynamic compositions to Java uses anonymous classes, which creates automatically a closure with the inner elements. In the translation to  $C^\sharp$ , a component class has to be declared, and the names imported by the composition have to be declared as parameters of the constructor. The component class is no longer a singleton because each component value represents a different closure and therefore a different component.

**Boxing of primitive types** Another feature of the calculus ported to  $CC^\sharp$  is the parametric polymorphism in the components. In  $CC^\sharp$ , parametric polymorphism is extended to the interface types and to the fact that primitive types are allowed as type arguments. Implementing bounded type parameters makes use of a generalization of type variables to the bounding type (in unbound cases, to the `object` type), and of boxing/unboxing of primitive types. The automatic boxing/unboxing feature of .NET acts as an advantage, for it simplifies the implementation of the compiler in this aspect. On the other hand, foreseen developments in the CLR in this direction will greatly improve this implementation both in clarity and efficiency of the compiled code [10].

## 4.2 Type safe dynamic loading

An important feature of our component model is the use of component types under a discipline of structural type equivalence together with subtyping. In  $CC^\sharp$ , equivalence and subtyping for interfaces follow the standard .NET name equivalence, however we implement the original structural relationship between component types. This allows for a greater flexibility with respect to substitutability of component values; subtyping for component types is defined as expected. Roughly, a component type is considered a subtype of another component type if it declares more provided services, less required services, and the ports are subtyped covariantly for provided services and contravariantly for required services.

Execution errors resulting from incompatibilities between components (architectural mismatch errors) are statically detected by the type system. However, in an implementation supporting dynamic loading of components this kind of errors can only be detected at loading time of the component. Therefore, our runtime systems must be able to dynamically check the type of the loaded components, which is unknown at compilation time, and compare it with an expected static type, determined at compilation time. This isolates aspects of component loading from composition and instantiation and allows for a proper error handling in localized parts of the code.

To keep type information apart from the code we find it convenient to represent the type of components using CLR attributes. In this way, a loader is able to retrieve the type of components and match it against the expected type. This verification is performed by code generated by the  $CC^\sharp$  compiler and enforces a safety boundary between the “internal” well-typed environment and the “external” environment. In Fig. 8 we illustrate the translation of a  $CC^\sharp$  component to  $C^\sharp$  using a custom attribute (`CCSTypeAttribute`).

<pre>component A {   requires I r;   provides J p;   ... }</pre>	<pre>[CCSTypeAttribute("{r:I}=&gt;{p:J}")] class A : CCSComponent {   ... }</pre>
--	---

Figure 8: Translation of a  $CC^\sharp$  component type

The modularization of the representation of the type information allows future expansions on other loading verifications. Other component features could be represented using this approach, namely non-functional requirements like: remote location, parallel execution, fault tolerance, etc.

### 4.3 Assemblies and components

In .NET the unit of sharing and reuse is the assembly. Individual assemblies may contain classes as well as other resources packaged together with the metadata relevant to its elements. In the current compiler design, the compilation of a  $CC^\sharp$  source file generates an assembly where one or more  $CC^\sharp$  components and associated types are grouped together.

To support separate compilation of  $CC^\sharp$  components in the .NET environment it is required that the interface types used to specify the components are shared between the components and the client code. In .NET, sharing a type means to compile both parts against the same shared assembly. Which leads us to two possibilities. If the client code is compiled against the component, the interface types may be included in the component assembly with the component code. On the other hand, If the client application is developed independently, the client code and component can be compiled relying on an common assembly containing the specification of the types they share.

## 5 Summary

In this paper, we related the composition mechanisms of the component calculus proposed in [20] with features of the .NET run-time. In particular, we discussed the mapping between the component model and the object model of the Common Language Run-time (CLR), supporting component composition and parametric polymorphism. We have also addressed the use of .NET

attributes to ensure the type safe loading of components, and the mapping between components, types and assemblies.

Building on this work, we are developing a compiler for a glue language that can express architectural designs in an readable and statically verifiable way. Future developments on the mapping of parametric polymorphism can also be anticipated when features like generic type become available in the CLR. Reconfiguration of active component instances is also an issue of ongoing research, still in its early stages. We expect to make use of the explicit representation of the interconnection architecture of components present in our model to define and verify, by means of an appropriate type system, the consistency of reconfiguration actions.

**Acknowledgments** I want to acknowledge my PhD supervisor Luís Caires for many useful discussions. This work is partially funded by a Microsoft Research Grant - Contract No. 2002-73 (Component Glue).

## References

- [1] Jonathan Aldrich, Craig Chambers, and David Notkin. Archjava: Connecting software architecture to implementation. In *Proceedings of the International Conference on Software Engineering (CSE)*, 2002.
- [2] Davide Ancona and Elena Zucca. True modules for Java-like languages. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 2001.
- [3] Kim B. Bruce, Luca Cardelli, and Benjamin C. Pierce. Comparing object encodings. In *Proceedings of the Conference on Theoretical Aspects of Computer Software (TACS)*, 1997.
- [4] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. Modula-3 report (revised). Technical report, Digital Systems Research Center, 1989.
- [5] Matthew Flatt and Matthias Felleisen. Units: Cool modules for HOT languages. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, 1998.
- [6] I. R. Forman, M. H. Conner, S. H. Danforth, and L. K. Raper. Ira Forman, “Release-to-Release Binary Compatibility in SOM”. In *Proceedings of the Conference on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, 1995.
- [7] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Proceedings of the ACM SIGPLAN SIGACT Symposium on Principles of Programming Languages (POPL)*, 1994.

- [8] Atsushi Igarishi, Benjamin Pierce, and Philip Wadler. Featherweight java: A minimal core calculus for Java and GJ. In *Proceedings of the Conference on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, November 1999.
- [9] Andrew Kennedy, Nick Benton, and Claudio Russo. SML.NET: Functional programming on .NET. Talk in the second Microsoft .NET Crash Course for Faculty and PhDs, 2002.
- [10] Andrew Kennedy and Don Syme. The design and implementation of generics for the .net common language runtime. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, 2001.
- [11] Dave MacQueen. Adaptation in hot languages: Comparing polymorphism, modules, and objects. In Tony Hoare, Manfred Broy, and Ralf Steinbruggen, editors, *Proceedings of the NATO Advanced Study Institute on Engineering Theories of Software Construction, August 2000*. IOS Press, Marktobendorf, Germany, 2001.
- [12] 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.
- [13] Hsieh W. McDirmid S., Flatt M. Jiazzi: New-age components for old-fashioned java. In *Proceedings of the Conference on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, October 2001.
- [14] Erik Meijer and Clemens Szyperski. What’s in a name: .NET as a component framework. In *Online proceedings of the First OOPSLA Workshop on Language Mechanisms for Programming Software Components*, 2001.
- [15] John Mitchell, Sigurd Meldal, and Neel Madhav. An extension of Standard ML modules with subtyping and inheritance. In *Proceedings of the ACM SIGPLAN SIGACT Symposium on Principles of Programming Languages (POPL)*, Orlando, FL, 1991.
- [16] Benjamin C. Pierce. Advanced module systems (a guide for the perplexed). Invited talk on ICFP 2000, slides available online from the URL <http://www.cis.upenn.edu/~bcpierce/papers/>.
- [17] Andreas Polze. Component programming. Talk in the first Microsoft .NET Crash Course for Faculty and PhDs, 2001.
- [18] Clemens Szyperski Rosziati Ibrahim. The COMEL language. Technical report, School of Computing Science, Queensland University of Technology Brisbane, Australia, 1998.
- [19] João Costa Seco. Adding type safety to component-oriented programming. In *First FMOODS PhD Student Workshop*, 2002.

- [20] João Costa Seco and Luís Caires. A basic model of typed components. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 2000.
- [21] João Costa Seco and Luís Caires. Parametrically typed components. In *WCOP'2000, Fifth Workshop on Component Oriented Programming*, 2000.
- [22] Vugranam C. Sreedhar. Mixin'up components. In *Proceedings of the International Conference on Software Engineering (CSE)*, Orlando, Florida, USA, May 2002.
- [23] Wolfgang Weck. Do we need inheritance? In *Workshop on Composability Issues in Object-Oriented Programming at ECOOP'96*, Linz, Austria, 1996.
- [24] Matthias Zenger. Type-safe prototype-based component evolution. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Malaga, Spain, June 2002.