The .NET Primitives for Open, Dynamic and Reflective Component Frameworks

Mircea Trofin[1], Nicholas Blumhardt[2], and Clemens Szyperski[1]

[1] Microsoft Corporation
{mirceat,clemens}@microsoft.com
[2] Readify
nicholas.blumhardt@readify.net

Abstract. "Composition Primitives" is a novel component model targeting .NET. The model facilitates composition across component programming frameworks via an adaptation mechanism external to the component. Constructing adapters is relatively inexpensive, because the model is minimal and focused on just one concern: offering enough information to support composition. Although small, the model supports static discovery of the services provided and consumed by a component—in other words, it is reflective. To strengthen the value of its reflection capabilities, it purposely does not rely on the Service Locator pattern and it supports n–order composition scenarios. In this paper, we present our model and support our claims.

1 Introduction

We present a component model that serves as a foundation for creating Open, Dynamic .NET applications built out of Reflective components. These components may have been developed in a domain–specific programming model, or may have been developed in a different, possibly legacy, component framework.

We will use the acronym "ODR" for these kinds of applications. In ODR applications, third–party functionality (components) can be added or removed (the "open" quality), possibly while the application is running (the "dynamic" quality), and there are first–class means to statically determine, for such third–party functionality, what facilities it provides and what its requirements are (the "reflective" quality). Reflection typically needs to be performed without loading any component code to avoid associated performance penalties. Eager loading tends to become prohibitive in applications with a large number of components that do not need to be all loaded upfront—like an integrated development environment or a web browser.

Examples of existing component frameworks targeting the construction of ODR applications include: CORBA Component Model [1], Castle MicroKernel/Windsor [2], Autofac [3], and Fractal [4].

In our model we focus on discovery and basic composition. We believe this is the core concern of any component framework, and, therefore, other aspects commonly covered by component frameworks can be realized separately or built on top of our model. We designed the Managed Extensibility Framework (MEF), part of .NET 4.0, as one such more comprehensive framework, validating our layering.

In particular, a characteristic property of components is that they form units of versioning [6]. Like other higher-level concerns, the primitives do not address versioning directly. Instead, it is left to component frameworks built on top of the primitives to address such concerns appropriately. While the details are beyond the scope of this paper, MEF, for instance, relies on the underlying versioning semantics in .NET and supports adapters to deal with additional versioning issues.

In contrast to other models targeting ODR applications, we claim our model offers the following novel and differentiating capabilities:

Our model supports creating domain-specific programming models and facilitates composition across component frameworks. A "programming model", in this context, is syntax and semantics defining components[1]. Since .NET is a multi–language platform, the term "syntax" refers to both language–specific, as well as language–independent means of expression. An extreme example of the former would be a language supporting the keyword `component`, with a compiler targeting the .NET Common Instruction Language (CIL) specification [5]. An example of the latter is the use of generally–supported .NET concepts, like custom attributes, types, or properties, to define what a component is, regardless of language. For example: any type annotated with the custom attribute `Component` is a component.

Choosing a component framework is an architectural decision [6]. It is hard to move away from such decisions: As applications evolve, it becomes important to enable interoperability with components written for other component frameworks—perhaps more recent ones. It is always possible to enable this—one can always write custom adapters and wrap components on a case-by-case basis. This tends to be expensive. We offer a solution for some of the most repetitive problems, such as discovery, without using the classical solution of a Service Locator.

The Service Locator pattern, also known as the Lookup pattern [7], is a widely used mechanism for late binding in open systems. It consists of a naming service, where service providers register under a name (typically a string). Consumers of services are bootstrapped to the naming service and use such a name (obtained through bootstrapping or as a parameter) to imperatively find and utilize a service.

Similar to the designers of other frameworks for ODR applications [1,3,2], we see the Service Locator pattern as hindering reflectivity and opted for a solution pertaining to the alternative pattern, namely Inversion-of-Control [8].

Supporting domain-specific programming models, or other component frameworks, is easy and inexpensive, from an engineering perspective, because the model is small and focused just on core concerns. As cost is one of

[1] This definition makes the term "programming model" synonymous to "component model". We intentionally use the term "programming model" to indicate layering with respect to the Composition Primitives, i.e. a component model adapted to the Primitives.

the primary concerns of any engineering team, ensuring the model imposes a small cost is inherently important.

Our model supports n-order composition without imposing requirements on the component author. Complex applications take complex dependencies—such as dependencies on providers of services (other components) rather than just the services themselves. We refer to these as n-order dependencies, and we will define them in more detail later in the paper. It will be explained that it is important to facilitate discovery of these kinds of dependencies without forcing component developers into modeling for each such concern—i.e. a solution and guidance is necessary in the framework, rather than be left to component and application authors to decide upon.

The remainder of this paper is organized as follows: We provide an outline of our model to offer support for our claims in the next section. A broader comparison and contrast with related work in the area of component-oriented software as well as other technologies follows. We conclude with an outline of current applications of our model in commercial and in open-source projects that validate the applicability of our solution.

Other literature in the area of component frameworks uses mathematical formalism to introduce and prove properties of the respective component framework [4]; we believe that, for our purposes in this paper, using a well-established programming language (and its semantics) is sufficient. Given that our model targets the .NET platform, we use C$^{\#}$. For compactness and enhanced readability, we removed unnecessary (for this paper) annotations like visibility keywords ("public") or, in some places, type casts. As such, we assume basic familiarity with C$^{\#}$and .NET, but provide an appendix with an overview of less familiar features that our model relies on, such as lambda expressions, delay–compiled expressions, and the functional model of sets (features commonly known as "Language Integrated Query", or LINQ, which were introduced to the scientific community as *Cω* [9] and have been part of C$^{\#}$since .NET 3.5).

2 The Primitives Model

The Primitives model (see figure 1) consists of: Services, Service Definitions, Instances, Components, and Dependencies[2].

Somewhat different from other component frameworks, ours is a management model, meaning that instances of the Primitives model are separate from operational program instances, used only for composition, and can be discarded after composition occurred, thus imposing no runtime overhead once composition or re-composition completes.

[2] This ontology happens to translate into the .NET Managed Extensibility Framework (MEF) model, which builds on the Primitives, using different names: Export, ExportDefinition, ComposablePart, ComposablePartDefinition, and ImportDefinition. The paper uses the Service, Instance, Component terminology. Refer to `http://mef.codeplex.com` for more information on MEF, including the open-source implementation and samples.

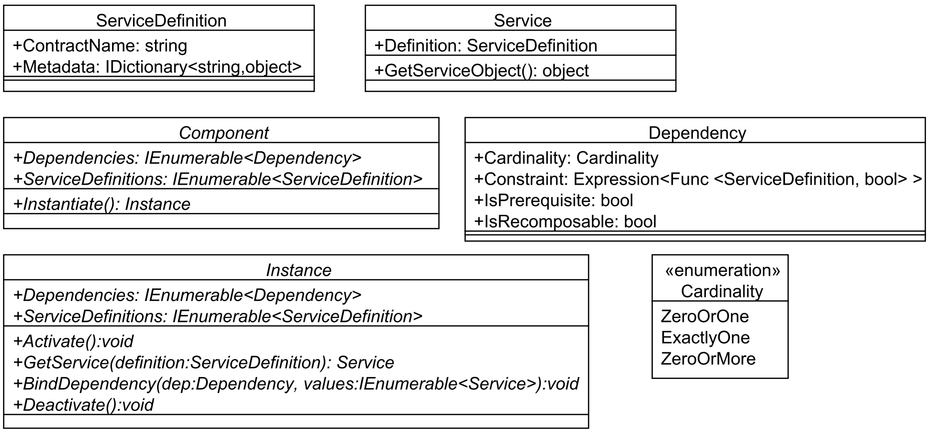
ServiceDefinition
+ContractName: string
+Metadata: IDictionary<string,object>
Service
+Definition: ServiceDefinition
+GetServiceObject(): object
Component
+Dependencies: IEnumerable<Dependency>
+ServiceDefinitions: IEnumerable<ServiceDefinition>
+Instantiate(): Instance
Dependency
+Cardinality: Cardinality
+Constraint: Expression<Func <ServiceDefinition, bool> >
+IsPrerequisite: bool
+IsRecomposable: bool
Instance
+Dependencies: IEnumerable<Dependency>
+ServiceDefinitions: IEnumerable<ServiceDefinition>
+Activate():void
+GetService(definition:ServiceDefinition): Service
+BindDependency(dep:Dependency, values:IEnumerable<Service>):void
+Deactivate():void
«enumeration» Cardinality
ZeroOrOne
ExactlyOne
ZeroOrMore

Fig. 1. Overview of the Primitives Model

A Service Definition is a semi–structured data object. It describes a kind of functionality that is offered for utilization, or consumption. It is the kind of information that one would use to decide whether a Service is interesting, without requiring the loading of code, which is a feature desirable in ODR applications.

A Service Definition is modeled as an atomic and opaque contract name (a string) and a string–keyed dictionary of objects, referred to as "metadata". The contract name identifies a document in the general sense, i.e., information, referred to as "contract", that describes the functionality offered for consumption. This description may be parameterized, in which case the contract must also describe how to encode such parameters using the metadata part of the Service Definition.

Let us consider an extensible stack-based calculator application—a perhaps contrived example, but suitable for illustrative purposes. The calculator can be extended with new operators. The contract for a particular operator may use, for the contract name, the string `Operator`, which implies the requirement that metadata contain the key `Symbol` and an associate string-typed value, such as "+".

A Service is an association between a mechanism (`GetServiceObject`) for obtaining an object satisfying a Service Definition, and the Service Definition. This is the basic building block for the rest of the model and it represents functionality ready-to-use by a consumer. Consumers may expect that the object obtained via `GetServiceObject` respect whatever prescriptions the contract requires—typically, that it implements some interface.

A Component represents a unit of reusable code. The code may be used if its dependencies are satisfied. Since dependencies may be satisfied with different values, an Instance (discussed further below) represents a particular such satisfaction which can be used independent from others.

A component advertises the set of Dependencies that all its Instances will need to have satisfied (the `Dependencies` property), and the set of statically–known Service Definitions all these Instances will offer (the `ServiceDefinitions` property).

Components produce Instances via the `Instantiate` method. Separate calls to this method must always result in different Instances being produced. Such Instances may end up offering more Services than statically advertized by a Component.

To enable composition, a given component's dependencies need to be analysed and candidate components must be determined that could satisfy these dependencies. Candidates whose own dependencies cannot be satisfied in the given composition context need to rejected. Component analysis and selection/rejection may occur without loading any code—based solely on the information provided by its `ServiceDefinitions` and `Dependencies` properties. Code loading can be deferred to the point a first request for utilizing a Service offered by an Instance is made. Compared to systems that load component code early, composition analyses and resulting selection/rejection decisions that are performed without loading component-specific code can have significant performance advantages, as validated by our use of MEF in Visual Studio 2010.

An Instance represents a set of Services and a set of Dependencies. Over time, more or less Services may be available, depending, for example, on the satisfaction of some optional Dependencies, or other runtime conditions. Mandatory Dependencies ("prerequisites") need to be satisfied before an Instance can be asked for Services.

To satisfy a Dependency, an external agent calls `BindDependency` and provides a set of Services. The agent may use `GetService` to retrieve a particular Service that is offered by the Instance.

Once all dependencies are satisfied, the Instance is ready to be used, meaning any of its offered Services may be utilized. In particular, this is the time an implementation may decide to load the actual component code and perform the actual satisfaction of dependencies.

Dynamic composition or recomposition is supported by allowing Instance dependencies be rebound on a live Instance.

Finally, a Dependency is defined through a `Constraint`. This is an Expression object applicable on a Service Definition and producing a boolean (a filter, essentially). Expressions are typed, delay–compiled functions. In our context, they are assumed to be pure. For more information, refer to Appendix A.

The key feature of using .NET expressions for the Constraint is that we can describe arbitrarily complex boolean expressions, with terms that may be arbitrary, type–safe navigations in the structure of the metadata of a Service Definition. We will discuss how this is the basis for supporting complex composition scenarios (section 2.5), as well as how this simplifies the composition process (section 2.2), when compared to known alternatives used in related work (section 3).

Other concepts exposed on a Dependency, but not explored in this paper, are: cardinality of the Dependency (`ZeroOrOne`, `ExactlyOne`, or `ZeroOrMore`); whether it is prerequisite (a generalization of the notion of constructor parameter); and whether if it can be rebound (support for dynamic scenarios).

An example of a Constraint would be the expression, using C$^{\#}$lambda notation:

```
(sd)=>sd.ContractName=="Operator" &&
        sd.Metadata.ContainsKey("Symbol") &&
        sd.Metadata["Symbol"]=="+"
```

This Constraint expresses a dependency on services respecting the "Operator" contract, and, in particular, on those that have a metadata key called `Symbol`.

The calculator in our example could expose itself as a component with two dependencies: one on a stack, with cardinality `ExactlyOne`, and one on a collection of Operators, with cardinality `ZeroOrMore`. The constraints of these dependencies would be as follows:

```
//stack
(sd)=>sd.ContractName=="Stack";

//any operator
(sd)=>sd.ContractName=="Operator" &&
        sd.Metadata.ContainsKey("Symbol")
```

In the case of a dependency on operators, the only statically known data is that the contract name needs to have a particular value, and that the metadata must contain the "Symbol" key—our calculator will use this for user interaction purposes, to identify each operator. In the case of a dynamic dependency (one that is satisfied based on dynamic system behavior), the Constraint should still be expressed in terms of statically–known properties of Service Definitions, and the Cardinality should always be `ZeroOrMore`. With that, it is possible to reason about composition of even dynamic dependencies—and still before loading component code.

As a final note about the Primitives, it is not necessary that Services be produced by Instances, nor that Instances be produced by Components. For example, a Service representing external functionality (e.g. a web service) may simply be instantiated by the application and subsequently considered for composition just like Services produced by Components. An implementation provided by a host that must be shared as a singleton by all hosted components may be represented as an Instance. This is also the reason that both Services and Instances carry their descriptions (i.e. Service Definitions and Dependencies)—in order to allow for analysis in the absence of a Component object.

2.1 Simple Example

We use the calculator example to illustrate the Primitives model. To avoid noise, in this example, the programming model used *is* the Primitives, however, in practice this is atypical - the Primitives are meant to be implemented by adapters to programming models, while the components would be implemented in such programming models.

Figure 2 shows the implementation of Component, and Figure 3 for the corresponding Instance for a generic operator. The example assumed a few constructors for some of the Primitives, which we have previously excluded from the model for brevity. In Figure 2, line 10, a new Service Definition is constructed with the contract name "Operator" and the metadata being a dictionary with only one key value pair, the key being "Symbol" and the value the variable `symbol`. Line 12 assumes a constructor for Dependency that constructs a constraint on a contract name ("Stack" in this case), and assumes an `ExactlyOne`

```
class OperatorComponent:Component{
    internal Func<double,double,double> Op{get;set;}
    private ServiceDefinition _offeredSvc;
    private Dependency _dep;

    OperatorComponent  (string symbol, Func<double,double,double> f){
        Op = f;
        _offeredSvc = new ServiceDefinition(
                "Operator", {"Symbol",symbol}));
        _dep = new Dependency("Stack");
    }
    override IEnumerable<ServiceDefinition> ServiceDefinitions{
            get { yield _offeredSvc; }
    }
    override IEnumerable<Dependency> Dependencies{
        get { yield _dep; }
    }
    public override Instance Instantiate(){
        return new OperatorInstance(this);
    }
}
```

Fig. 2. Example Operator Component

```
 1  class OperatorInstance : Instance
 2  {
 3    private OperatorComponennt _theComp;
 4    private Stack<double> _stack;
 5    private Service _theService;
 6
 7    OperatorInstance(OperatorComponent comp){
 8      _theComp = comp;
 9    }
10    override IEnumerable<ServiceDefinition> ServiceDefinitions{
11      get { return _theComp.ServiceDefinitions; }
12    }
13    override IEnumerable<Dependencies> Dependencies{
14      get { return _theComp.Dependencies; }
15    }
16    override void BindDependency (Dependency dep, IEnumerable<Service> values){
17        var stackExp = values.First();
18        _stack = (Stack<double>)(stackExp.GetServiceObject());
19    }
20    override Service GetService(ServiceDefinition svcDef){
21      if (svcDef.Name == "Operator"){
22        if (_theService == null){
23          Func<double> operator = () => _theComp.Op(Stack.Pop(), Stack.Pop());
24          _theService =  new Service(svcDef, () => operator);
25        }
26        return _theService;
27      }
28      return null;
29    }
30  }
```

Fig. 3. Example Operator Instance

cardinality. In Figure 3, line 24, we construct a Service object based on a given service definition, and where `GetServiceObject` delegates to the provided function (in this case, a function returning the `operator` function).

In this example, one would obtain the component for the "+" operator as follows:

```
Component plus=new OperatorComponent("+",(x,y)=>x + y)
```

As it can be seen, the "Operator" contract stipulates that the service object have the type `Func<double,double,double>`, and the "Stack" contract requires a `Stack<double>` object. This illustrates the fact that the Primitives do not impose any limitations over the kinds of types that may constitute valid operational interfaces to services. Concretely, `Func<>` is a sealed type, while `Stack<>` may be inherited from (both are part of the core .NET Framework).

Because the Service Definitions used are pure data, one can reason over a space of such Service Definitions using the Constraint of a Dependency and, without loading any component code, make determinations over feasibility of composition.

2.2 Composition in the Primitives Model

Composition in the context of the primitives consists of creating Instances out of Components and resolving their Dependencies with Services obtained out of other Instances. The composition may be controlled by an agent, generally referred to as "composition engine", external to the components involved. The composition engine is expected to satisfy a dependency `d` with a Service `s` for which the following expression evaluates to true (where `Compile` is a standard method on .NET expression objects—see Appendix A):

```
d.Constraint.Compile()(s.Definition)
```

Our goal for domain independent composition is supported by the fact that the engine need not understand the constraint in order to determine whether a service may be used to satisfy dependencies. Parsing the Constraint is possible (since it is an Expression) and may be useful for optimizations, such as indexing. It is also useful if the engine is capable of recognizing and treating specially particular kinds of contracts.

A characteristic of the Primitives design is that it has no built–in notion of identity. In particular, this allows for defining new Components from existing Components through an equivalent of the notion of partial application found in functional languages. A custom implementation of Component can be constructed that, based on an existing Component and a set of pairs (`Dependency`, `IEnumerable<Service>`), presents itself as exposing the same Service Definitions as the original component, and the same Dependencies, except for those provided in the set of pairs. Calls to Instantiate lead to the creation of instances of the original component, where the Dependencies provided in the set are hidden and pre–satisfied by the values they were associated with.

2.3 Supporting Other Component Frameworks and Domain–Specific Programming Models

We implemented an extensible chatting application[3]. Components for this application can be developed using the attributed programming model that is provided as reference implementation with .NET Framework 4.0, or as "plain old CLR" (POCO) types typically composed by Autofac[3]. In turn, these components may be composed using either the composition engine that is part of .NET Framework 4.0, or with the Autofac container. The former was designed upfront to be based on our model, while the latter was adapted to support our model.

The ChatClient.ManagedExtensibilityComponents project comprises MEF components, while the ChatClient.AutofacComponents project comprises POCO types for Autofac.

[3] `http://nblumhardt.com/archives/composition-primitives/`

The ChatClient.ManagedExtensibilityHost project uses the MEF CompositionContainer and PocoAdapter types to host both MEF and POCO components.

The ChatClient.AutofacHost project hosts the same components, but uses the Autofac.Integration.CompositionPrimitives adapters to host both MEF and POCO components in the same (Autofac) container.

2.4 Cost of Constructing Adapters

The adapter that allows Autofac components to be composed by the .NET composition engine operating on our model can be seen in the PocoAdapter project of the chatting application. The adapter consists of an implementation of the Component and Instance concepts. This adapter totals 125 lines of code, supporting our claim of low-cost adapter construction.

2.5 Modeling Higher–Order Dependencies

In our model, a dependency is taken on a Service, via a condition over the description of that Service. Typically, that Service represents a value with meaning in the application domain, and no meaning in the Primitives model. It is possible, however, that we represent a Component, for example, as a Service: after all, it provides the service of generating Instances. In this case, the value represented by the Service has a meaning in the Primitives model, which is understood and utilized by its consumer. We call this a higher–order dependency.

Scenarios where higher–order dependencies are required are apparent in complex composite applications, where some components act as generic containers for other components, managing their life–cycle and controlling access to them. An example would be an Integrated Development Environment (IDE) that can be extended with custom designers for Graphical User Interfaces (GUIs). Such designers may be usable as independent applications (i.e. they can be used as a component), however, internally, such designers may be themselves extended, and may use other components (like an editor, a canvas, etc). Hence, designers are higher-order components that also compose (and expose services of) other components.

To support such kinds of dependencies, one option would be to manufacture contracts that make the high–order dependency implicit, for example, a dependency on a component that produces an “Operator” would be encoded as a dependency on a Service respecting the contract “Operator Component”. The problem with this approach is that it does not scale—the set of such contracts is a power set of the set of “simple” service contracts.

The options we discuss in what follows express higher–order dependencies through a description of structure, and require composers to understand a few contracts, one for each kind of higher–order dependency. Our goal for programming model–independent composition is still supported, however, components exposing such higher-order dependencies end up being less reusable than “simple” components. Still, any composition engine would be able to assess at least that it is unable to compose them.

In this paper, we will only illustrate support for dependencies on components.

Dependencies on Components. Consider a scenario where we want to package the wiring of a given stack with a private instance of a calculator user interface component and a private instance of each of the operator components available in some set of components. Then, we want to treat this package as a component.

Next, suppose we want to create a new Component, where each Instance would internally instantiate two Calculator aggregates and connect them to a shared instance of a Stack. The dependency of this new Component would be solely on other components: a Stack Component, a set of Operator Components, and an Aggregate Component (as defined above). It can be observed that this mechanism allows for recursive definitions of "Components".

To model such dependencies, we introduce a contract named "Component" which requires that the describing ServiceDefinition have a metadata property called "ServiceDefinitions", which should be a collection of ServiceDefinitions. Said metadata property maps to the corresponding property of a Component. A Service exposing the "Component" contract produces a Component when `GetServiceObject` is called. All that is required is that a composition engine understand the contract name "Component" and map its set of available Components into a set of Services correspondingly.

A dependency on the addition operator component we used in our examples so far would be expressed through a constraint as follows (where `Any` is the .NET LINQ operator for existential quantification over an enumerable; also, some type casts removed for brevity):

```
(sd)=>sd.ContractName=="Component" &&
        sd.Metadata.ContainsKey("ServiceDefinitions") &&
        sd.Metadata["ServiceDefinitions"] is
                IEnumerable<ServiceDefinition> &&
        sd.Metadata["ServiceDefinitions"].Any(
                s=>s.ContractName=="Operator" &&
                     s.Metadata.ContainsKey("Symbol") &&
                     s.Metadata["Language"]=="+")
```

This illustrates the rationale for using expressions for describing constraints in dependencies: they are sufficiently powerful to describe the conditions the Service Definition needs to exhibit, yet, since they represent delay–compiled code, a composer need not interpret their contents. The composer simply needs to search the expression tree for the equality test between the Service Definition contract name and the string `Component`, then compile the whole expression (via the `Compile` method expression objects expose in .NET) and evaluate the resulting function over the Service Description representation of currently available components.

3 Related Work

The area of component–based programming [6] features a large number of component models. This Section will only focus and contrast a relevant sample of those that can be used to build ODR applications. For instance, models that focus on static ahead-of-time composition, such as Koala [10], are not discussed.

The two traditional areas supported by ODR component models are extensible enterprise and rich-client applications. Representative for the enterprise area are the Enterprise JavaBeans (EJB) framework [11], the CORBA Component Model (CCM) [1], and more recently, Dependency Injection containers. Rich-client applications use frameworks such as COM [12]. Besides these, there are several frameworks that developed out of research, such as Fractal [4]. In the following, we contrast these frameworks and our model.

A component in EJB is a bundle consisting of a lifecycle manager (“home”, optional as of EJB 3.x), component code, and a deployment descriptor (additional EJB–specific information). Offered services are modeled as Java interfaces, and composition is driven by component code, by using the Java Naming and Directory Interface (JNDI) [13]—a Service Locator mechanism. Only component homes can be found and bound via JNDI, component instances are subsequently obtained via the component home.

EJB features a hybrid between Service Locator–based mechanisms and explicit dependency declaration. The home of a component “A” is registered at deployment time with a server–wide name (e.g. “Server–A”). Another component, “B”, wishing to use “A”, uses a relative name (e.g. “B–A”) when using JNDI programatically to lookup “A”. Then, the relative name and the expected home interface are listed, by the component developer of B, in the deployment descriptor. Finally, at deployment time, the application composer associates the relative name to the server–wide name.

This mechanism may be considered a way to expose component dependencies, however, nothing stops a component from attempting (and succeeding at) a lookup using a “guessed” server–wide name. As such, knowledge of an EJB component’s dependencies is generally incomplete, when such knowledge is based solely on information supported by the component model. Given the introspectable nature of Java bytecode, it is possible to construct tools to extract complete inter–component dependency information (e.g. [14]), however, this is something the component model supports accidentally, and the mechanisms required may not be suitable for all applications, since they require time–consuming bytecode parsing, and since some EJB containers generate merged container/component code at deployment time.

CCM is an extension of CORBA, and aims at allowing the creation and composition of components developed on a variety of operating system platforms and using different languages. In CCM, a component features “ports”, describing: (i) implemented IDL interfaces (“facets”), (ii) required implementations of IDL interfaces (“receptacles”), (iii) produced and consumed events (“sources” and “sinks”), and (iv) “attributes”, which are configurable properties. Except for attributes, which are intended to be primitive types (e.g. integers, strings), the ports are expressed in terms of IDL interfaces. For example, a receptacle is expressed in terms of the interface the component wishes to consume, as well as an optional indication on whether this is rather a collection of such interfaces. Just like in our model, and in contrast to EJB, a CCM component does not rely on a Service Locator to be composed—composition is externally managed.

Components themselves are described using IDL. In contrast to our approach, IDL describes the operational interface of the consumed service, not the service itself. For example, all calculator operations in our example would implement the same IDL interface. However, the operation that they implement would not be part of that interface, while it would be an interesting criterion for expressing dependencies. Our model provides the notions of Contract, as separate from operational interface, and Metadata for this purpose.

The closest to our Service Definition concept is the concept of WSDL [15] documents in the area of Web Services. WSDL, however, is heavily Web–oriented. Conceptually, both utilize semi–structured data to describe a service. In the same area, the Component Object Model framework (COM) [12] uses the Windows registry as basis for component discovery. The information that may be stored in the registry is similar conceptually to the notion of Service Definitions. Similarly, the enterprise-oriented COM+ uses a registration database to store component metadata.

The Vienna Framework [16] has similar inter-component framework composability goals to our effort. It takes the approach of "wrapping" components, thus imposing a constant runtime penalty. A second effect of this approach is a larger solution space, with a model trying to address the different operational interface features that are currently popular - e.g. methods, properties, events. More importantly, it does not have a model for dependencies. The metadata concept in Vienna revolves around operational contracts, while our concept of metadata models a broader kind of contracts - similar, as noted, to the Web Services notion of contracts.

A number of Dependency Injection frameworks describe components in terms of offered and required interfaces. Examples include the Castle Microkernel [2] and Autofac [3]. Both target the enterprise space, and both use a conventions–based approach to comprehending a plain .NET class as a component. For example, constructor parameters are treated as dependencies, and their type is used for matching with the types offered by other components. In turn, the interfaces implemented by a type are considered as offered services. There are policies for matching requirements expressed on collections, or inheritance–based matching. While both these frameworks use plain .NET types to model components, since the translation between a type and the internal notion of a component differs, interoperability is hard to achieve.

Outside of the enterprise space, the Fractal component model [4] was designed explicitly to permit language–specific implementations, albeit without the interoperation goal that CCM has. It relies on Service Locators to drive binding to other components, through a binding controller, which a component may expose. Binding controllers expose offered interfaces, but do not expose requirements, which is one contrasting difference to our model. Fractal models explicitly components that aggregate other components, through the concept of a content controller, which, if implemented by a component, it allows for other components to be added or removed from it.

The capability offered via content controllers is simlar to the capability, in the Primitives model, to model dependencies on components (see Section 2.5). The main difference is that the Primitives do not need a separate concept to achieve the same result, in fact, the concept used (the Dependency model) is reusable for a variety of other kinds of dependencies, as was illustrated in section 2.

OSGi [17], the component model foundation of systems like Eclipse, addresses features similar to our model. Despite its layered architecture, and unlike our Primitives model, OSGi has a large surface area with complex protocols around aspects like deactivation and reactivation. Primitives relegate such protocols to the space of normal contracts and expects that, in the context of specific systems like MEF, such contracts are published and well-known. There is no expectation that such protocols are necessarily used across all systems build on top of Primitives. As a result, it is much easier to build implementations or adapters to and from our model. As a further point of distinction: OSGi uses LDAP query expressions to describe constraints. While LDAP is a broadly accepted directory access standard, it was not designed for the specific needs of a higher-order component model. Our model draws on a general-purpose delay-compiled and type-safe expression model (a standard part of the .NET framework) that we apply over a general metadata design, to support higher-order composition directly.

The mechanism used to express dependencies is somewhat reminiscent of the way design–by–contract languages Eiffel [18], ESC/Java [19], or SpecSharp [20] specify pre– or post–conditions by using a rich expression language describing Boolean constraints.

The concept of decoupling the concern of modeling components from the concern of composing them is found in the area of Architecture Description Languages (ADL). An ADL is a formal language that describes how components are instantiated and connected. Typically, ADLs are used for architectural validation. Interesting examples, in the context of this paper, are the Darwin ADL and ArchJava, a Java extension that can express architectural constraints.

Darwin[21,22] introduced the notion of hierarchical composition—which is similar to the kind of second order composition the Primitives support (Section 2.5).

ArchJava [23] is an ADL extension to the Java language, mainly aimed at validating communication integrity. It offers keywords for defining components with ports (`component`, `port`). A port specifies method signatures that it provides and that it requires (the `provides` and `requires` keywords) . A `connect` keyword can be used to compose components, by connecting `provides` and `requires` port elements of various components. Like Darwin, ArchJava also supports hierarchical composition through the concept of "composite components". Unlike our model, and similar to CCM, ArchJava does not specify a notion of contract separate from operational interface contracts.

4 Conclusion

We have presented a component model for building reflective components for open and dynamic applications in .NET. Using a publicly–available sample chat application, we showed how the model supports arbitrary programming models and domain-specific component frameworks. Using this example, we showed how the development effort required to adapt such programming models or component frameworks to our model is relatively small.

A key feature of our model is the utilization of the .NET delay–compiled code model, or expressions, for describing dependencies. This offers support for higher–order dependencies, without significant complexity on the side of composition engines, and without requiring that component authors be aware of such potential consumption scenarios.

The model has current practical applications. It forms the foundation for the Managed Extensibility Framework (MEF), which is part of the .NET Framework 4.0[4]. In turn, MEF is used by Visual Studion 2010 as both an internal component model and as an external third-party extensibility mechanism. Other implementations of the primitive model have been contributed by third-parties[5], for example, an IronRuby[6] programming model[7].

References

1. Object Management Group. Corba Component Model (2002), `http://www.omg.org`
2. The Castle Project. Microkernel, `http://www.castleproject.org`
3. Nicholas Blumhardt. Autofac, `http://www.autofac.org`
4. Bruneton, E., Coupaye, T., Leclercq, M., Quema, V., Stefani, J.-B.: An open component model and its support in java. In: Crnković, I., Stafford, J.A., Schmidt, H.W., Wallnau, K. (eds.) CBSE 2004. LNCS, vol. 3054, pp. 7–22. Springer, Heidelberg (2004)
5. ECMA International. Standard ECMA-335 - Common Language Infrastructure (CLI), 4 edn. (June 2006)
6. Szyperski, C., Gruntz, D., Murer, S.: Component Software: Beyond Object-Oriented Programming, 2nd edn. Component Software. Addison-Wesley/ACM Press (2002)
7. Kircher, M., Jain, P.: Pattern-Oriented Software Architecture. In: Patterns for Resource Management, vol. 3. Wiley, Chichester (2004)
8. Johnson, R.E., Foote, B.: Designing reusable classes. Journal of Object-Oriented Programming 1(2), 22–35 (1988)
9. Black, A.P. (ed.): ECOOP 2005. LNCS, vol. 3586. Springer, Heidelberg (2005)
10. van Ommering, R.: Software reuse in product populations. IEEE Transactions on Software Engineering 31(7), 537–550 (2005)
11. Sun Microsystems. Java 2^{TM} Platform Enterprise Edition Specification, v1.4 (November 2003), `http://java.sun.com/j2ee/j2ee-1_4-fr-spec.pdf`
12. Box, D.: Essential COM. Addison-Wesley, Reading (1998)
13. Lee, R., Seligman, S.: The Jndi API Tutorial and Reference: Building Directory-Enabled Java Applications. Addison-Wesley Longman Publishing Co., Inc., Boston (2000)
14. Trofin, M., Murphy, J.: Static verification of component composition in contextual composition frameworks. International Journal on Software Tools for Technology Transfer 10(3), 247–261 (2008)

[4] For reference, the open source code is available at `http://mef.codeplex.com`

[5] Available at `http://www.codeplex.com/MEFContrib`

[6] IronRuby is an open source implementation of Ruby for .NET. More information is available at `http://ironruby.net/`

[7] Used in `http://www.mahtweets.com/` Twitter client

15. World Wide Web Consortium (W3C). Web services description language (WSDL) v. 2.0, http://www.w3.org/TR/wsdl20
16. Oberleitner, J., Gschwind, T., Jazayeri, M.: The vienna component framework enabling composition across component models. In: ICSE 2003: Proceedings of the 25th International Conference on Software Engineering, Washington, DC, USA, pp. 25–35. IEEE Computer Society, Los Alamitos (2003)
17. OSGi Alliance. Osgi service platform – release 4, version 4.2 (2009)
18. Meyer, B.: Eiffel: The Language. Prentice-Hall, Englewood Cliffs (1992)
19. Flanagan, C., Leino, K., Lillibridge, M., Nelson, C., Saxe, J., Stata, R.: Extended Static Checking for Java. In: Proceedings of Programming Language Design and Implementation (2002)
20. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# Programming System: An Overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 49–69. Springer, Heidelberg (2005)
21. Ng, K., Kramer, J.: Automated support for distributed software design. In: CASE 1995: Proceedings of the Seventh International Workshop on Computer-Aided Software Engineering, Washington, DC, USA, p. 381. IEEE Computer Society, Los Alamitos (1995)
22. Magee, J., Dulay, N., Eisenbach, S., Kramer, J.: Specifying Distributed Software Architectures. In: Botella, P., Schäfer, W. (eds.) ESEC 1995. LNCS, vol. 989. Springer, Heidelberg (1995)
23. Aldrich, J., Chambers, C., Notkin, D.: Archjava: connecting software architecture to implementation. In: ICSE 2002: Proceedings of the 24th International Conference on Software Engineering, pp. 187–197. ACM, New York (2002)

A .NET Framework Features

The model relies on two features in the .NET framework, which, for the purpose of clarity, are briefly described in what follows. These features are: type–checked delay–compiled code, also known as "Lambda Expressions" (in short, "expressions" in this paper), and a monadic functional model for collections, commonly referred to as "LINQ" (Language–INtegrated Query).

An expression is an object of type `Expression<Func <...>>`, and may be constructed in two ways: either through explicit construction of an abstract syntax tree, by using typed nodes, or through language facilities, in languages that support that. For example, C#provides syntax and compile–time verification for constructing such objects. The following line of code constructs the object that represents the division computation of two integers, returning a double:

```
Expression<Func<int,int,double>>
        divisionExpression = (x,y)=>x/y;
```

Expressions can be used two ways. One way is to compile them, which results in a typed function ("delegate" in .NET nomenclature), which can then be applied:

```
Func<int,int,double> division =
    divisionExpression.Compile();

double result = division(1,2);
```

Alternatively, expressions can be passed to interpreters. LINQ–to–SQL, for example, translates an expression to SQL statement, which is then evaluated by a relational database server.

The second feature we mentioned is LINQ, which allows for the definition of lazily–enumerated collections in a functional style. For example, given a collection of integers `integers`, the following defines the subset of even numbers.

```
IEnumerable<int> evens =
    integers.Where(i=>i%2==0);
```

Besides the `Where` operator, Linq defines a large number of further standard operators over IEnumerables: functions from enumerable to enumerable. In combination, these can be used to express a full range of queries. Some languages, such as C$^{\#}$, provide syntactic sugar for a large subset of these operators. For example, the following form yields the same even numbers:

```
IEnumerable<int> evens =
    from i in integers where i%2 == 0 select i;
```

The rich support of expressions and enumerator operators, C$^{\#}$language sugar, and dynamic compilation, in combination, create a potent foundation for the constraint system of the Composition Primitives presented in this article.