Masters Thesis

A component model for the .NET CLR

by

Tobias Rickli

Due date
21. September 2010

Advisors:
Jan S. Rellermeyer
Ioana Giurgiu
Prof. Gustavo Alonso

ETH Zurich, Systems Group
Department of Computer Science
8092 Zurich, Switzerland
Abstract

The OSGi framework is a module system and service platform for dynamically deploying services into networked environments in Java. Though initially this technology targeted home service gateways, nowadays it is used as a mechanism to design and build extensible Java applications. Its growing adoption is mainly due to its support of a dynamic service deployment life-cycle and ability to support remote management. Microsoft’s .NET platform provides MEF, a library that addresses the design of extensible applications, but still lacks explicit support for building these dynamic systems like OSGi for Java allows. In this thesis, we analyze the challenges of implementing an OSGi-like component model for the .NET platform and how the CLR has to be extended to realize such an implementation, equivalent to OSGi. Differences between .NET and Java are analyzed and solutions are discussed to accommodate assembly-level modularization in .NET.
### 0.1 Glossary

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLR</td>
<td>Common Language Runtime</td>
</tr>
<tr>
<td>CLI</td>
<td>Common Language Infrastructure</td>
</tr>
<tr>
<td>ECMA</td>
<td>Ecma International; private non-profit standards organization</td>
</tr>
<tr>
<td>IL</td>
<td>Intermediate Language</td>
</tr>
<tr>
<td>JIT</td>
<td>Just-In-Time compilation</td>
</tr>
<tr>
<td>OSGi</td>
<td>OSGi Alliance (formerly 'Open Services Gateway initiative')</td>
</tr>
<tr>
<td>OSGinet</td>
<td>OSGi-like framework on the SSCLI 2.0 implementation of the .NET CLR</td>
</tr>
<tr>
<td>SSCLI</td>
<td>Shared Source Common Language Infrastructure; previously codenamed 'Rotor'</td>
</tr>
</tbody>
</table>
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1 Glossary</td>
<td>iv</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Motivation</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Goals</td>
<td>1</td>
</tr>
<tr>
<td>2 A comparison of Java and .NET</td>
<td>3</td>
</tr>
<tr>
<td>2.1 Packaging</td>
<td>3</td>
</tr>
<tr>
<td>2.2 Classloader</td>
<td>4</td>
</tr>
<tr>
<td>2.3 Search path and dependency resolving</td>
<td>5</td>
</tr>
<tr>
<td>2.4 Compile time type referencing</td>
<td>5</td>
</tr>
<tr>
<td>2.5 Independence of deployment unit and type name</td>
<td>5</td>
</tr>
<tr>
<td>3 Extending the CLR with modularity</td>
<td>7</td>
</tr>
<tr>
<td>3.1 Modularity</td>
<td>7</td>
</tr>
<tr>
<td>3.2 OSGi</td>
<td>7</td>
</tr>
<tr>
<td>3.3 Analysis of abstractions</td>
<td>8</td>
</tr>
<tr>
<td>3.3.1 Application Domains</td>
<td>8</td>
</tr>
<tr>
<td>3.3.2 Assemblies</td>
<td>9</td>
</tr>
<tr>
<td>3.4 Unloading assemblies</td>
<td>9</td>
</tr>
<tr>
<td>3.5 Assembly resolving</td>
<td>10</td>
</tr>
<tr>
<td>3.6 Interfaces</td>
<td>11</td>
</tr>
<tr>
<td>3.7 Versioning</td>
<td>11</td>
</tr>
<tr>
<td>4 Implementing the component model</td>
<td>13</td>
</tr>
<tr>
<td>4.1 Modules</td>
<td>13</td>
</tr>
<tr>
<td>4.2 Life-cycle management</td>
<td>14</td>
</tr>
<tr>
<td>4.3 Events</td>
<td>14</td>
</tr>
<tr>
<td>4.4 Services</td>
<td>15</td>
</tr>
<tr>
<td>4.5 OSGinet console</td>
<td>15</td>
</tr>
<tr>
<td>5 Choosing the CLR implementation</td>
<td>17</td>
</tr>
<tr>
<td>5.1 Shared Source Common Language Interface 2.0(SSCLI)</td>
<td>17</td>
</tr>
<tr>
<td>5.2 Mono</td>
<td>17</td>
</tr>
<tr>
<td>5.3 Portable.NET</td>
<td>18</td>
</tr>
<tr>
<td>6 Related work</td>
<td>19</td>
</tr>
<tr>
<td>6.1 Java OSGi</td>
<td>19</td>
</tr>
<tr>
<td>6.2 OSGi for .NET</td>
<td>19</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>6.3 Comparison to Inversion of Control and Dependency Injection</td>
<td>20</td>
</tr>
<tr>
<td>6.4 Comparison of OSGi and the Managed Extensibility Framework</td>
<td>20</td>
</tr>
<tr>
<td>6.4.1 Concepts in MEF</td>
<td>20</td>
</tr>
<tr>
<td>6.4.2 Granularity</td>
<td>21</td>
</tr>
<tr>
<td>6.4.3 Services</td>
<td>21</td>
</tr>
<tr>
<td>6.4.4 Versioning</td>
<td>21</td>
</tr>
<tr>
<td>7 Conclusions</td>
<td>23</td>
</tr>
<tr>
<td>8 Future work</td>
<td>25</td>
</tr>
<tr>
<td>A Example of a modular application on .NET</td>
<td>29</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Motivation

The increasing complexity of modern software demands approaches facilitating the reuse of software. Experience shows that modularity is a powerful model to separate functional units of software systems and reuse the componentized building blocks to a much higher degree. The OSGi Service Platform Specification[5] provides a framework for modularization, life-cycle management and dynamic deployment of services for networked environments. Although originally developed for home services gateways[6], OSGi has spread its influence to mobile and embedded devices[11], plug-in functionality for applications[1][7] and many other fields. The same functionality and modularization is desired for the .NET environment. First as extensibility, modularization and life-cycle management framework, ultimately, in an equivalent extent as R-OSGi including dynamic service deploying and remote management, distributed deployment and discovery of bundles[12].

1.2 Goals

The goal of this thesis is to implement the idea of modularity, lightweight service orientation, and life-cycle management for the .NET CLR. More specifically, the following issues need to be tackled:

- identify the mismatches between Java OSGi and .NET
- extend CLR with the missing functionality to support modularity
- implement an OSGi-like runtime
- evaluate a significant piece of software and draw conclusions on its performance

In a first phase, it has to be identified which operations and abstractions (e.g. Assemblies, Application Domains) the CLR supports, how these entities can communicate, and how these mechanisms could be used in an OSGi-like component model. In a second phase, the CLR has to be enhanced with the missing functionality. The third phase is to create a runtime comparable to
Introduction

OSGi that is able to handle components written in different languages supported by the CLR, and to evaluate a significant piece of software on it.

Differences between Java and .NET that influence the implementation of an OSGi-like framework, such as code loading and unloading, have to be analyzed and strategies to bridge the differences need to be discussed to better understand the necessary extensions on the CLR.

Mechanisms have to be identified to support the implementation of modularity and bundles such as assemblies and application domains. These can be used to represent modules and logical boundaries for code execution. The packaging of classes to JAR files and the representation of bundles through them are studied and differences to the .NET equivalents identified.

To extend the CLR to better accommodate an OSGi-like framework, an open implementation of the CLR is chosen that allows changes. The CLR has to be extended with functionality to closer emulate a Java-like behavior and facilitate the implementation of the framework.

The implementation of an OSGi-like component model is done on the extended CLR and uses the OSGi Service Platform Specification [5] for reference. The component model provides functionality for installing and uninstalling modules and a service registration/discovery.

To test and demonstrate the component model, an example application is implemented that makes use of bundle life-cycle management, service layer and event handling.
Chapter 2

A comparison of Java and .NET

In this chapter, we look at differences between the Java and .NET runtimes that are relevant for the implementation of a component model: packaging of modules, classloading, searchpath (classpath) for external resources, and dependency resolving of referenced resources are discussed.

2.1 Packaging

A typical Java application is packed in a JAR file that contains the directory tree and files of the application. The JAR file is an archive file type and is based on the ZIP format. Structuring is done by collecting classes to packages and representing this structure in the directory tree of the JAR file. An example of a JAR structure is shown in figure 2.1. This allows organization of the application structure and relevant resources like configuration files, graphics, data files, etc. can be integrated in a single file.

```plaintext
- JAR-file
  - main
    - ch
      - ethz
        - ClassFile.class
  - META-INF
    - MANIFEST.MF
  - OSGi-OPT
    - doc
    - src
```

Figure 2.1: Example of an OSGi bundle JAR structure

As shown in figure 2.2, a .NET application can consist of several files. Executables, as well as libraries are represented by assemblies that can be split
into separate files, called modules. The manifest, placed in one of the modules, describes the assembly and its corresponding files. Namespaces, the .NET equivalent to Java’s packages, are not represented in the file system.

![Figure 2.2: Assembly files](image)

### 2.2 Classloader

The classloader is the part of the runtime, that dynamically loads classes into the environment [8]. In Java there are an infinite number of possibilities how the classloader structure works. By default, there are three classloaders responsible for loading all classes in the runtime environment:

- The bootstrap class loader, responsible for loading core libraries in the /lib folder of the Java runtime.
- The extension class loader, responsible for loading libraries in the /lib/ext folder.
- The system class loader, responsible for loading all resources on the classpath.

The system classloader can be extended or replaced by a structure of one or more user-defined classloaders that work together by delegation. The user-defined classloaders all inherit from the `java.utilClassLoader` type. The smallest loadable amount of code is a class. Because of the highly customizable system of classloaders, it is possible to define a custom classloader for every module and enforcing access control for modules. OSGi does this by turning the hierarchical structure of the Java classloader system into a general graph[6], delegating the classloading to the bundle classloaders which enforce the access control by matching requests with the defined exports in the bundle manifest.

In the .NET CLR there is a class loader for every assembly. Types are loaded on-demand by the class loader. It is not possible to replace the class loader or change its behavior. Since the smallest loadable amount of code in .NET is an assembly, the classloader does not provide exactly the same service as in Java, but extracts the types from the in-memory representation of the assembly. Since the loading of classes can not be customized like in Java, it is impossible for an application to control loading of referenced assemblies, therefore difficult to create a dependency system with access control on the application level.
2.3 Search path and dependency resolving

By default, the Java runtime searches its classpath for class files. The classpath is defined by an environment variable, command-line argument or a system property. Because of Java’s flexible class loading system, it is possible to change the process of dependency resolving by defining custom class loaders, for example, to load classes over HTTP, which is important for remote deployment in R-OSGi[12], or dynamically change the classpath. To reference classes from external JAR files, they have to be placed in the classpath at runtime and imported in the code. Every class loader creates a separate namespace for classnames. Class user.Test loaded by ClassLoader1 is not the same as user.Test loaded by ClassLoader2.

In .NET, only the base directory of the application domain is searched for statically referenced assemblies. Additionally, it is possible to customize the search path in an application configuration file to include subdirectories of the base directory. It is not possible to point the search path above the base directory in the file system. This complicates dynamic deployment of bundles. Additionally, to achieve functionality, similar to [12], remote loading of bundles would have to be realized by means of a web service-like mechanism.

2.4 Compile time type referencing

In Java, class dependencies between types are not explicitly recorded at compile time but rather at runtime. Thus, the provider of the referenced type at runtime is not necessarily the same as referenced on compile time.

Not so in .NET, where references are on assembly level and are recorded at compile time. This increases assembly dependencies and leads to inflexibility in locating a reference on runtime[6].

2.5 Independence of deployment unit and type name

In Java, an imported type is not bound to its implementational representation. A type that is imported in a Java class is independent from its representation. It can be imported as a JAR into the project and later be supplied as a .class file or remotely loaded at runtime.

In .NET, the assembly name is part of the type name. This makes it impossible to resolve the type to a different implementation than was referenced at compile time. The dependency between assemblies is further increased. The type is only consistent if it is supplied in the assembly.
Chapter 3

Extending the CLR with modularity

3.1 Modularity

Modularity is a programming paradigm that enforces the principle of dividing a big problem into smaller problems and solving them separately, thus reducing complexity. An entity of modularization is called module. By employing modularity, complex systems can be built while keeping maintenance effort manageable. It reduces dependencies between logical parts of applications.

The modules communicate by providing interfaces to each other, by which interaction is defined. Modularity is described by cohesion and coupling, which usually correlate inversely. Cohesion describes the 'togetherness' of a module, how strongly the functionalities of a module are related. High cohesion usually increases the manageability and reusability of modules while decreasing dependencies. Coupling describes the grade of dependency of a module. The more a module has to rely on other modules to provide its functionality, the higher the coupling. A low amount of coupling is desirable in modular systems for the same reasons as high cohesion.

3.2 OSGi

The OSGi Service Platform (OSGi) is a dynamic module system for Java, with the OSGi framework as its core component. The OSGi Alliance, responsible for standardization and specification, publishes the OSGi Service Platform specifications, that are implemented by several institutions and companies, providing a range of available products.

In OSGi, modularity is reached by dividing a system into bundles, OSGi’s equivalent of a module. OSGi provides functionality for bundle life-cycle management, dependency management, service registration and discovery, and security, designed and arranged as layers as shown in figure 3.1.

The module layer is responsible for defining the implementation of modularity by specifying the notion of bundle and the specification of the content of the manifest file and its syntax, including bundle dependency by imports and
3.3 Analysis of abstractions

To accommodate a framework like OSGi on .NET, an important decision is how a component (bundle) is represented. Nothing like the Java packages exists in .NET that provides the same functionality. Either the component is represented on the level of an application domain, to gain the possibility of unloading it, but accepting the overhead of communication by .NET remoting, or an assembly is chosen as implementation for a bundle which leads to better performance, but makes unloading of the code impossible in the commercial .NET CLR. This trade-off is discussed in [6].

3.3.1 Application Domains

Application domains are the isolated environment where applications are executed. They are represented by the AppDomain objects. Each application domain has its own virtual address space and represents a boundary for security of executed code and unloading. Code execution across application domain boundaries is only possible by remoting, which creates a large overhead in comparison to normal function calls. Unloading an application domain unloads all containing assemblies, that are not loaded in a different application domain.
3.3.2 Assemblies

Assemblies represent an entity of compiled and packaged code in .NET. An assembly can either be an executable (.EXE) or a library (.DLL). A started executable is the root assembly of its application domain. Communication between assemblies of the same application domain is done with function calls and does not need serialization. Since we use a modifiable implementation of the .NET CLR (unlike [6]), the assembly is chosen as representation of a bundle and the framework extended to provide the necessary functionalities.

3.4 Unloading assemblies

In [15], three reasons are given, why Microsoft refrains from implementing a possibility to unload assemblies:

1. References to running code would have to be tracked on assembly granularity which is too expensive, compared to the current solution, which is application domain granularity.

2. A loaded assembly consists of its metadata, IL representation, and JITed code in the loader heap. The loader heap is not designed to free up memory dynamically and has to be changed to a malloc-style heap. Then all compiled methods would have to be tracked down and unloaded, which is expensive at development time and runtime.

3. Assemblies that are loaded in several application domains (shared assemblies) would have to be tracked and unloaded in all concerned application domains.

Ad 1: When referencing an already uninstalled bundle, it is acceptable to raise an exception and interrupt execution. Error handling is done by the OSGinet framework or the calling bundle.

Ad 2: On account of the complexity of unloading all JITed code, it is acceptable to only unload the metadata and IL representation of an assembly.

Ad 3: In the scenario of OSGinet, where all assemblies are used in only one application domain, shared assemblies can be neglected.

The loaded assemblies in the runtime are managed by the object representation of the application domain. Each application domain has a list of assembly objects and maintains a cached map, where assemblies are stored with their corresponding AssemblySpec as key. To discover, which data structures are involved in the process of assembly loading, the CLR was analyzed with the WinDbg debugger. To unload an assembly from an application domain, the assembly has to be removed from the list and the cache, to prevent a later reloading from the cache. Remove methods were added to ArrayList and AssemblySpecBindingCache for this purpose. Then the file handle of the assembly is freed and the assembly object deleted.

Generally, the CLR is not designed to release loaded assemblies. The best example is the usage of the class LoaderHeap throughout the CLR, which does not support reallocating freed memory.
3.5 Assembly resolving

There are two different possibilities for an assembly to load another assembly: by the `using` statement and compile time reference or by using the `System.Reflection` namespace and `Assembly.LoadFrom()` method.

To use types of imported bundles, the first option has to be used and the external assembly has to be referenced statically by its name. At runtime, this name is resolved to a specific assembly file by the assembly resolver, which then is loaded by the assembly loader. If nothing further is specified in the application configuration, the following files are searched for in this order [10]:

- application base directory/assemblyname.dll
- application base directory/assemblyname/assemblyname.dll
- application base directory/assemblyname.exe
- application base directory/assemblyname/assemblyname.exe

The second option circumvents the resolving process, by providing the path to the assembly file. But, since the necessary types were not referenced at compile time, reflection has to be used to call methods of the loaded assembly.

In the context of OSGi, the notion `resolving` describes a different process. To be allowed to actually load and run code, a bundle has to resolve its dependencies and `wire` all imports to corresponding exports [5].

In OSGiNet, bundle imports are resolved recursively in the `Start` method of the bundle, before the call to the `BundleActivator.Start` method. Since the assembly is also referenced at compile time, it would be loaded by the CLR anyway, if it is located in either the application base directory or a subdirectory with the same name as the assembly.

Because it is difficult to identify the currently loading assembly in the CLR, it was not possible to prevent the CLR from implicitly loading referenced assemblies. The attempt to disallow all assembly loading except root assembly
and system libraries failed. Only the root assembly and `mscorlib.dll` are specially identified in the application domain. The mechanism of calling an already loaded assembly could not be separated from the process of loading an assembly before the method call. The bundles were not able to make calls to the OSGinet framework any more.

This is part of the call stack when a referenced assembly is searched:

```
mscorwks!AppDomain::LoadDomainAssembly
mscorwks!Module::LoadAssembly+0x18a
mscorwks!Assembly::FindModuleByTypeRef+0x121
mscorwks!ClassLoader::LoadTypeDefOrRefThrowing+0xe4
```

It was not possible to identify the referenced assembly at any of these stages. Without this adaption, a bundle would be able to call methods from referenced bundles, regardless of its access specifications.

### 3.6 Interfaces

References from assembly A to types of assembly B are handled by the class-loader of the referencing assembly A, delegating the type loading to the class-loader of the referenced assembly B. Types of the same name but of different assemblies are not considered the same type. Therefore, an interface C included in assemblies A and B is not considered the same type in both assemblies. An implementation of the interface C in assembly B can not be cast to an object of the type C in assembly A. If that was allowed, it would be possible to extract types as services or bundle references, without statically referencing bundles at compile time, further reducing bundle dependency.

### 3.7 Versioning

One of the key features of OSGi is running multiple versions of a bundle simultaneously. The bundle version is located in the manifest file.

Running multiple versions of an assembly is usually not possible in .NET. A static reference to an assembly is resolved only by its name and does not take the assembly version into account. Therefore, loading a different version of an already loaded assembly will result in an exception.

To enable the CLR to distinguish between multiple versions, the assemblies have to have a strong name. The strong name consists of the name, version, and culture information of the assembly and a public key and signature of the assembly. With this tuple, the assembly is uniquely identified and can be loaded and run alongside other versions of this assembly. Also, it has to be referenced by its strong name to be loaded, i.e. if a strong named assembly was referenced at compile time, exactly the same assembly has to be present on runtime. To obtain a strong name, the developer has to create a key-pair with the `sn.exe` utility and provide the key at compile time. The compiler then signs the assembly.

Strong named assemblies can not reference weakly named assemblies, but only other strong named assemblies. In the context of OSGinet, every bundle assembly has to have a strong name to be able to interact with other bundles.
The difficulty here is that, since OSGinet is an assembly itself, bundles, compiled for a certain version of OSGinet, will only work with this version.

This influences the way, a bundle can be updated in .NET. The bundle would either have to be recompiled with the new version as reference or a rebind can be defined in the application configuration and the bundle restarted.
Chapter 4

Implementing the component model

OSGinet is an implementation of an OSGi-like component model on the .NET CLR. It includes functionality for modules, life-cycle management and services and provides functionality for bundle loading and unloading, registration and usage of services and notification based on events. The structure of OSGinet is based on the interfaces of [5] but deviates from them in many aspects, due to the differences between .NET and Java. To clarify the notions, a small example application is used.

4.1 Modules

Modules are represented by the class Bundle and consist of one assembly. The example application is divided into three bundles: Logger, AddressBook, and TestBundle.

Like in Java, it is possible to include attributes in the manifest file. In .NET, the attributes are part of the type system and have to inherit from Attribute in the System namespace. The attributes are limited and provide only enough configuration functionality to start a bundle and list imports and exports. Bundle imports and exports are defined by the BundleImportAttribute and BundleExportAttribute classes. To be able to execute the bundle, the bundle activator has to be identified in the BundleActivatorAttribute and has to implement the IBundleActivator interface.

These are the import and activator attributes of the AddressBook bundle:

[assembly: BundleActivator("AddressBook.AddressBookActivator")]
[assembly: BundleImport("C:\\Logger\\bin\\Logger.dll")]
4.2 Life-cycle management

By default, a bundle is loaded at the framework initialization to control bundles with the console. Here bundles can be installed, started, stopped and uninstalled. The console bundle uses the life-cycle methods of the BundleContext class. The OSGi net console itself is implemented as a bundle, started by default at framework initiation.

![Bundle states and transitions in OSGi](image)

Transitions between events are signaled by firing a bundle event with the according target bundle state.

4.3 Events

.NET provides functionality for events that can call registered methods. To be notified of state transitions, a bundle has to register an event handler with the target bundle. The target state is passed to the event handler by argument. By registering an event handler in the Logger bundle, the AddressBook bundle is notified, if the logger is not available any more:

```csharp
public void Start(Framework.BundleContext context)
{
    ServiceReference loggerService = context.GetService(
        typeof(ConsoleLogger).ToString());
    loggerService.Bundle.BundleEvent += Bundle_BundleEvent;

    context.RegisterService(typeof(IAddressBook).ToString(),
        new AddressBook((ILogger)loggerService.Service), null);
}

void Bundle_BundleEvent(object sender, BundleEventArgs e)
{
    if (e.State == BundleState.STOPPING)
    {
        Console.WriteLine("The logger bundle has stopped");
        ServiceReference service = context.GetService(
            typeof(ConsoleLogger).ToString());
        service.Bundle.BundleEvent -= Bundle_BundleEvent;
    }
}
```
4.4 Services

Since it is not possible in .NET to define events with delegates in interfaces, the additional abstraction with interfaces, as it is used in Java OSGi, is left away and bundles are referenced by the `Bundle` type throughout the framework. Events should be unregistered by the bundle that registered it, latest in the `Stop` method of the activator.

4.4 Services

To reduce dependencies between bundles, functionality can be shared over a service layer. Services can be registered by name, searched for by name and will fire events on state changes. A service consists of an object in the bundle that is registered as the service and a name, usually the typename of the registered object or its interface. In figure 4.3, the registered services of the example bundles are shown. Services have to be unregistered by the supplying bundle.

To get a service from the registry a bundle has to know the name of the service. Usually services are registered and discovered by the full typename of the service interface. To call methods on the service object, it is cast to its interface type.

![Service Objects](image)

Figure 4.3: Registered service objects

4.5 OSGinet console

To manage bundles, the OSGinet console is loaded in the initialization of the framework as a default bundle. The console bundle can be left away to use the framework without user interface. The following commands are available:

- `install` (bundle file) installs a bundle by its full path
- `start` (bundle id) starts a bundle by its bundle id. The bundle is resolved and its dependencies installed and started, before the start method of the BundleActivator is called.
- `stop` (bundle id) if the bundle is in the ACTIVE state, the stop method of the BundleActivator is called and the bundle state set to RESOLVED.
- `uninstall` (bundle id) stops the bundle if it was in ACTIVE state, unloads the assembly and removes it from the framework
- `stat` shows the list of installed bundles with their current state and the bundle location in full path.
Implementing the component model
Chapter 5

Choosing the CLR implementation

Three different implementations of the CLR were taken into consideration. To decide which open implementation should be used, completeness, ease of use, complexity of implementation and documentation were looked at.

5.1 Shared Source Common Language Interface 2.0 (SSCLI)

The Shared Source Common Language Interface 2.0 is Microsoft’s source distribution of the .NET 2.0 CLR. It is published under a non-commercial shared source license which allows modification and redistribution for educational and personal use.

It includes compilers for C# and JavaScript, development tools, build tools, test suites, etc. Not included are e.g., the ASP.NET framework or Windows Forms.

Unlike the commercial .NET distribution, the SSCLI can be run and built on FreeBSD and Mac OS X as well as Windows XP SP2 (but not on other versions of Windows). This is possible because of the Platform Abstraction Layer (PAL). The PAL translates system calls to its respective target operating system API.

The source code in the SSCLI 2.0 distribution is not the same as the source code of the commercial .NET 2.0 CLR, it is however based on an early branch of the .NET 2.0 source tree.

This implementation of the CLR was chosen because it is close to the commercial version. Unfortunately, the project was discontinued by Microsoft and therefore, a lot of resources and documentation are not available any more.

5.2 Mono

Mono is an open source implementation of the .NET CLR. It implements the ECMA standards for C# and the CLI and consists of compiler, runtime, and base class library. Non-ECMA-standard libraries are implemented in the mono class library.
Mono can be run on several Unixes, Linux, and Windows. ASP.NET applications can be ported to mono, several GUI toolkits are supported, and development of WebServices is possible as well.

Due to its size and complexity, Mono was not chosen as basis for this thesis.

5.3 Portable.NET

Portable.NET is part of the DotGNU project and, like Mono, implements the ECMA standards for C# and CLI. The project aims at 100% compatibility with Microsoft’s .NET base class library and provides an implementation for the non-ECMA-standardized Windows.Forms namespace.

The latest (2007) release is Portable.NET 0.8.0 which still misses some functionality and therefore was not chosen as basis of this thesis.
Chapter 6

Related work

6.1 Java OSGi

OSGi is a widespread solution for modularization problems for many years. The OSGi Service Platform Specification [5] is used as reference. There are several implementations of the framework that are specialized on particular tasks.

Concierge [11][4], for example, is an implementation of the OSGi Specifications R3 and specializes on mobile and embedded devices and provides a GUI management interface. It has a 80kB footprint and is specially optimized for compact frameworks.

Equinox[1] is a certified implementation of the OSGi R4 core framework specification and part of the Eclipse project. Equinox is an all-purpose implementation that is used in the Eclipse IDE.

Apache Felix [2] is an OSGi implementation under the Apache license using OSGi principles for a dynamic deployment framework. A concern of the Apache Felix project is to create a platform, that is compatible with the other OSGi implementations and complies strongly to the OSGi specification. The Apache Felix project contains many subprojects that extend the frameworks functionality.

Knopflerfish [3] is an open source implementation of the OSGi Specification R4 v4.2. It fully implements the framework and the OSGi Compendium Services. It is an open source spin-off from one of the founding members of the OSGi Alliance, Gatespace.

6.2 OSGi for .NET

In [6], results are presented of an effort to create an OSGi-like service platform for the .NET platform. The paper is about differences in code loading between Java and .NET and how the OSGi specification extends the code loading of Java. Differences in granularity of code loading, dependency recording at compile time, class search order and service provider substitutability are shown.

Four approaches to implementing an OSGi-like platform are discussed. The single application domain approach is limited because of the inability to unload assemblies in a single application domain. The multiple application domain approach allows the unloading of bundles by unloading the complete applica-
Communication overhead is criticized since communication between application domain requires .NET remoting. The third approach uses the shared domain to supply executed application domains with service interfaces. Although internal CLR functionality could be used for service discovery, the communication overhead through remoting and serialization remains.

The fourth approach suggests a hybrid of the first and second. Highly coupled bundles are loaded into one application domain to profit from local method invocation. Each application domain would have a service registry which delegates non-local service requests to a global service registry. This would allow existence of multiple versions of bundles in different application domains.

The paper concludes on the inability of assembly unloading and overhead of inter-application domain communication.

This paper was used as groundwork for this thesis. The focus on assembly unloading and assembly resolution is a result of its findings.

6.3 Comparison to Inversion of Control and Dependency Injection

OSGi is often compared to the concepts of Dependency Injection (DI) and Inversion of Control (IoC). IoC is a design principle that circumvents the flow control of procedural programming. The principle is best described by the famous Hollywood quotation "Don’t call us, we’ll call you!". Methods are not called sequentially, but usually registered or delegated to a framework. DI is the pattern that uses IoC for reducing module dependency by delegating object creation and discovery to a framework.

Although OSGi includes functionality that is capable of implementing these principles (service registry/discovery), its focus lies on modularization and lifecycle management, distinguishing it from IoC containers.

6.4 Comparison of OSGi and the Managed Extensibility Framework

In the current release of the .NET CLR (4.0), the Managed Extensibility Framework (MEF) is introduced as a new library that simplifies the design and creation of extensible applications and components1. Its main target is to create applications that can be extended by third-party developers and to provide a plug-in-like design.

6.4.1 Concepts in MEF

A part provides services to other parts and consumes services provided by other parts. Parts in MEF can come from anywhere, from within the application or externally; from an MEF perspective, it makes no difference.

Export An export is a service that a part provides. When a part provides an export, it is said that the part exports it. For example, a part may export a

1http://mef.codeplex.com
logger, or in the case of Visual Studio, an editor extension. Parts can provide multiple exports, though most parts provide a single export.

**Import** An import is a service that a part consumes. When a part consumes an import, the part imports it. Parts can import single services, such as the logger, or import multiple services, such as an editor extension.

**Contracts** A contract is an identifier for an export or an import. An exporter specifies a contract that it provides, and an importer specifies the contract that it needs. MEF derives contract names from the types that are being exported and imported.

**Composition** Parts are composed by MEF, which instantiates them and then matches up exporters to importers.

The concepts in the MEF are similar to those in OSGi with different nomenclature. The composable part corresponds to a bundle, contracts to wires, and composition to bundle resolving.

Although MEF and OSGi both provide a way to modularity and decoupling, there are several differences:

### 6.4.2 Granularity

In MEF the unit of modularization is a *composable part* that defines imports and export on the level of methods and fields but also on the level of whole classes.

In OSGi a bundle consists of a complete JAR file that can contain several classes and interfaces. Imports are limited to packages.

The granularity of modularization shows the intended use of the respective frameworks. Where MEF specializes on extensible applications, OSGi is used in a bigger picture to simplify the design and development of complex systems and to control the life-cycle of modules.

### 6.4.3 Services

In MEF there is nothing equivalent to the service layer of OSGi. Comparable to the OSGi service registry would be the discovery mechanism of MEF that is able to search folders of assemblies for a corresponding contract.

### 6.4.4 Versioning

Whereas OSGi allows running multiple versions of a bundle simultaneously, MEF does not provide a mechanism of versioning, other than assembly versioning. In the composition process, the imports are matched to exports without concern of version numbers.
Chapter 7

Conclusions

The first issue was to decide on which implementation of the CLR the work would be done. Mono, Portable.NET and the Shared Source CLI 2.0 where investigated with the central concern of implementing an unload method for assemblies. Since all three implementations were comparable in complexity, SSCLI was chosen as it is the closest to the commercial .NET CLR and provided the most complete functionality.

In all three implementations the loading of assemblies is a complex process and gaining insight as to reverse this process was cumbersome. The first approach to tackle assembly unloading in SSCLI was done by studying the source code, this did not yield enough information to extract all involved data structures.

After managing to build and run the SSCLI, a test application was built to test assembly loading in single and multiple application domains. As described in [6], using multiple application domains leads to communication and serialization overhead, therefore a single application domain approach was chosen.

While extending the test application to the OSGinet framework, the WinDbg debugger was used to gain insight in call stacks and data structures, involved in assembly loading. The AssemblySpecBindingCache in the assemblyspec files, and the ArrayList in the AppDomain class were identified as main data structures for assembly object storing. The UnloadAssembly method was added as an internal method of the AppDomain object, not, as proposed in [15], as a method of Assembly, to clarify the relationship between application domain and its assemblies.

The method had to be added to the C# implementation of AppDomain, marked as internal call, registered in the ecall class, responsible for handling internal calls to the CLR, and implemented in the appdomainnative class.

A difficulty was getting a reference to the application domain and the assembly that has to be unloaded. In the appdomainnative class, where all internal calls from the base library to the AppDomain are implemented, the references to assemblies and application domains are not handled as normal classes, but as AssemblyBaseObject and AppDomainBaseObject which are part of the object class. From these references, a pointer to the correct assembly and application domain can be extracted.

A simple delete of the AssemblyBaseObject or the underlying assembly object leads to access violation exceptions. Removing the complete list of loaded assemblies and replacing it with a copy without the respective assembly was
again not possible for the same reason as removing the single element. Since both the `ArrayList` and `AssemblySpecBindingCache` are implemented on a `LoaderHeap` the memory could not be simply freed but the elements had to be removed from within the data structure.

The `AssemblySpecBindingCache` is based on a hash map for pointer, called `PtrHashMap`, which provides the method `DeleteValue` to remove an entry. The key of the hash map is the hash of the assembly’s `AssemblySpec`. The assembly spec is not referenced by the assembly itself and a newly created spec would not provide the same hash value. Iterating over the map allowed to identify the entry before the call to `DeleteValue` and the assembly spec can be extracted from the iterator.

The `ArrayList` of assemblies in the application domain is traversed by iterator and the corresponding index found by comparing the assembly pointers. Then it is possible to assign `null` to the corresponding entry in the list.

An attempt on freeing JIT compiled code from the CLR by iterating over the method table resulted in a corrupt state of the execution environment.

The second issue worked on was transferring access control over assembly references to the OSGinet application. To reach control over assembly resolving, loading of assemblies was restricted to references from the root assembly in the application domain.

This was possible but prevented non-system and non-root assemblies not only from loading referenced assemblies but also from calling referenced assemblies. The mechanism of calling an already loaded assembly could not be separated from the process of loading an assembly before the method call. The problem of transferring access control to the OSGinet framework was not solved.

The OSGinet platform was extended to allow bundles to install, uninstall, start and stop, services to be registered, discovered and unregistered. Extensive configuration and filtering of attributes and services as in [5] was not implemented.

The evaluation of an application on OSGinet was replaced by a small example application to show the achieved functionalities of OSGinet.

An analysis of OSGi and Java shows, that OSGi heavily relies on specific details of the Java runtime, that are not provided by .NET. Namely the unloading of bundles, a dynamic class loader to provide access control over bundles and dynamically configurable search path. To implement a component model with the same applications as OSGi, the .NET CLR has to be enhanced to correspond to the needs of such a model. Although modularity on assembly level is possible in .NET, the design of the CLR contains mismatches with the needs of OSGi. To provide a complete life-cycle management, it has to be possible to remove assemblies from the CLR and update modules without restarting the application. This is reached by isolating the representation of an assembly, removing it from the application domain and deleting it. In OSGi, access control between bundles is managed by resolving imports and exports. To reach the same control in .NET, control over assembly resolving has to be transferred to the component model.
Chapter 8

Future work

The implementation of OSGinet was used to analyze the concepts of a component model on .NET. To reach the same functionality as a Java OSGi implementation, extensions to the .NET CLR and OSGinet have to be applied. The following issues provide pointers for future work:

- Extend the configurability of OSGi in the assembly manifest by adding attribute classes.
- Implement filtering of attributes and services.
- Transfer control over assembly resolving to OSGinet.
- Adapt type system to decouple referencing assemblies.
- Extend the CLR to take versioning of weakly named assemblies into account.
- Extend OSGinet to provide R-OSGi-like functionality [12].

The configurability of OSGinet can be extended by adding Attribute classes to the system for the bundle headers described in [5, page 34].

OSGi implements a LDAP-like filtering for its manifest file configuration [5, page 37]. By adding a filtering mechanism to OSGinet, a much higher precision of service discovery and attributes is possible.

To transfer control over assembly resolving to OSGinet, non-system assemblies should only be allowed to access assemblies that are already loaded by the root assembly, this being the OSGinet framework.

The references between assemblies are statical, making an update of bundles without application restart difficult. This can be solved by not resolving assembly references explicitly at compile time, but at runtime, thus allowing dynamical updates of bundles.

Version management of bundles is one of OSGi’s big advantages. To reach the same goal in .NET, the version has to be used as distinguishing mark when resolving weakly named assemblies.

A major support to spreading OSGi was extending it to a distributed platform. To reach this functionality, .NET would have to be extended to load code remotely.
Bibliography

Appendix A

Example of a modular application on .NET

To show how bundles work together, an example application is implemented, where three bundles communicate through the service layer.

The TestBundle adds and removes addresses to and from the AddressBook which in turn logs its activities with the Logger. When TestBundle is started, its dependencies are resolved and AddressBook is installed, which leads to the loading of Logger. The dependencies are defined in the respective assembly attributes; from AssemblyInfo of the address book assembly:

```csharp
[assembly: BundleActivator("AddressBook.AddressBookActivator")]
[assembly: BundleImport("C:\Logger\bin\Logger.dll")]
```

The Logger provides two services, a FileLogger and a ConsoleLogger and registers them with their class name. From LoggerActivator:

```csharp
public void Start(Framework.BundleContext context)
{
    context.RegisterService(typeof(ConsoleLogger).ToString(), 
        new ConsoleLogger(), null);
    context.RegisterService(typeof(FileLogger).ToString(), 
        new FileLogger("C:\log.txt"), null);
}
```
To use the logger services, `AddressBookActivator` looks them up and registers its own services in turn. Additionally, an event handler is registered, to be notified of changes in the logger bundle:

```csharp
public void Start(Framework.BundleContext context)
{
    context = context;
    loggerService = context.GetService(typeof(ConsoleLogger).ToString());
    loggerService.Bundle.BundleEvent += new Bundle.BundleEventHandler(Bundle_BundleEvent);

    context.RegisterService(typeof(IAddressBook).ToString(),
    new AddressBook((ILogger)loggerService.Service), null);
}

void Bundle_BundleEvent(object sender, BundleEventArgs e)
{
    if (e.State == BundleState.STOPPING)
    {
        Console.WriteLine("The logger service has stopped");
        loggerService = null;
    }
}
```

On the retrieved implementation of the `ILogger` interface, the methods are called to do logging on the console.

The following code of `TestBundleActivator` uses all three bundles:

```csharp
public void Start(Framework.BundleContext context)
{
    IDictionary<string, string> props = new Dictionary<string, string>();
    props.Add("lastname", "Rickli");
    props.Add("firstname", "Tobias");
    props.Add("street", "Musterstrasse 43");
    props.Add("zip", "8004");
    props.Add("city", "Zuerich");

    Address add = new Address(props);
    book.Add(add);
}
```

In the screenshot A.2, the management of the example application with the OSGi console is shown. After installing and starting `testbundle.dll`, the `stat` command shows all loaded bundles with their state and all registered services. Uninstalling the `Logger` bundle raises a bundle event that the `AddressBook` bundle subscribed to. The logger services are not available any more.
Figure A.2: Screenshot of the example application in use