Integration Models in a .NET Plug-in Framework*

Reinhard Wolfinger, Herbert Prähofer

Christian Doppler Laboratory for Automated Software Engineering
Johannes Kepler University, 4040 Linz, Austria
wolfinger@ase.jku.at
herbert.praehofer@jku.at

Abstract: Applications based on plug-in architectures are extensible through third-parties and enable customized user environments. We argue that extensibility and customization are important features in enterprise application software. In an ongoing research project we are developing a plug-in platform in .NET for the enterprise domain. Targeting the enterprise domain raises special requirements with security, reliability and versioning. This paper presents models for host and plug-in integration that address execution of plug-ins in reliable settings and allowing independent evolution of core applications and plug-ins and it will show how this has been solved in a consistent and transparent way in the .NET plug-in framework.

1 Introduction

The plug-in approach [Bg03] enables developers to build application software that is inherently extensible and customizable to the particular needs of an individual user. A small core application can be extended with features implemented as components that are plugged into the core and integrate seamlessly with it. Although plug-ins originally became popular with web browser applications like Mozilla Firefox [Sa00] or with the Eclipse Platform [Ec03], we have found good reasons to enable extensibility and customizability in application software for the enterprise space.

Extensibility is important because it is simply not possible for an application of any size or complexity to hit 100% of the requirements out of the box. Even if an application covers all major scenarios, the customer is always asking for more features. Some of those features are niche or industry specific. Or they may not be in the manufacturer's core competence, or they are far enough of the list in terms of priority. Thus for commercial reasons the manufacturer decides not to include those features in its base version of the product. However, the customer still needs a way to fill the gap between what the applic-

* This work has been conducted in cooperation with BMD Systemhaus GmbH, Austria, and has been supported by the Christian Doppler Forschungsgesellschaft, Austria.

ation provides and what they really need to satisfy their business. When extensibility is built into the product, this enables the customer or distributors to fill the gap between what the manufacturer is providing and what the customer really needs to solve its business problems. And in doing so they create a virtue cycle. The manufacturer can still sell the base version of the product and hit what he thinks is the majority of customer scenarios and allows third-parties to fill the gap. They create those specific plug-ins that make the application more able to participate in a wider range of customer scenarios and get even more deeply integrated in the customer's business processes. The application plus the wide range of plug-ins that are created on behalf of the manufacturer by third-parties result in an aggregate application, that has broader appeal to a wider range of customers.

Moreover, customizability is a critical factor for usability because enterprise software is inherently complex and feature-rich. Modern enterprises define a significant number of business processes, whereat the individual user participates only in a rather small fraction of those. Hence if the user interface of an enterprise application is cramped with features for all business processes, users struggle to find their share of features that they will use to participate in their own processes. The plug-in approach enables customization and therefore improves focus and reduces clutter by providing a customized user environment.

In an ongoing project we develop a new plug-in framework in .NET for the enterprise domain. Our plug-in framework shows much resemblance to Eclipse, however, whereas Eclipse describes extensibility with dedicated XML configuration files, our framework relies on .NET concepts such as custom attributes and metadata to specify relevant information directly in the source code of an application [Wo06]. We argue that this approach is more readable and easier to maintain. Moreover, it exploits .NET specific features for plug-in deployment, discovery, qualification, integration, activation, deactivation and unloading of extensions at run-time. Additionally our plug-in framework encounters special requirements with security, reliability and versioning stemming from the enterprise domain. In this paper we address reliability and versioning issues.

1.1 Problem Areas

The basic issues of a plug-in framework are simple. There is a host application that allows architectural changes through component addition, removal and replacement. To make that work, we have essentially seven problems to solve:

(a) Contract specification. The host defines a contract how it wants to be extended and how it intends to use a plug-in. The plug-in relies on this contract and has to provide an appropriate implementation.

(b) Plug-in deployment. The plug-in contributor needs to package the extension implementation and get it to the machine where it is going to run.

(c) Plug-in discovery. The plug-in framework needs to discover that a plug-in has been added. It is desirable that a host discovers plug-in additions at start-up as well as at run-time.

(d) Plug-in qualification. The plug-in framework needs to make sure that a plug-in is actually appropriate for using with this host.

(e) Plug-in integration. The plug-in framework has to integrate the plug-in, possibly in a lazy loading scenario where the plug-in is integrated without actually loading the code.

(f) Plug-in activation. When finally used, the plug-in framework has to activate the plug-in. Usually it will load the code and create the respective objects.

(g) Plug-in deactivation. The plug-in framework removes a plug-in that is no longer needed. Removal means to deactivate the instance, close the communication path, free the environment, unload the implementation and reclaim resources.

1.2 Special Requirements

As our framework targets enterprise applications instead of tool environments, new requirements emerge. Before we present our plug-in solution, we want to elaborate on a couple of challenges that we consider crucial for application of the plug-in framework in the enterprise domain:

(a) Reliability. In the enterprise domain customers expect high availability. Therefore, the host might need to take precautions against being taken down by a ragged plug-in. For example, if a plug-in will do calls to unmanaged code or other non-verifiable, insecure operations, the host must make provisions that it is executed independently and can protect itself from a crash of its plug-ins.

(b) Versioning. In an open plug-in world it is impossible to test all plug-ins against new versions of a host. If we want to update a host and still use the existing plug-ins or update a plug-in independently of the host, we need an architecture which is version resilient. A version resilient model should provide backward as well as forward compatibility. Backward compatibility means that a new host can load plug-ins that targeted earlier versions of the host. Forward compatibility means that an old host can load plug-ins that targeted newer versions of the host.

(c) Security. One important reason for enabling extensibility is to enable third parties to contribute functionality. A third party is typically a partner company, a reseller or even the user itself. The developer of the host application might probably restrict what a plug-in can do and can not do. The host application might give a plug-in some permissions but not all. And the plug-in developer might require certain permissions. The framework needs to match those permissions and requirements and to negotiate settings acceptable to both.

This paper will specially address the issues of executing plug-ins in reliable settings and allowing independent evolution of core applications and plug-ins. And it will show how this has been solved in a consistent and transparent way in the .NET plug-in framework. Security, however, is not addressed in this paper, but is subject to ongoing research. We will briefly discuss security in the outlook in the last section.

1.3 Outline of Paper

The content of this paper will be as follows: The next section reviews the basic concepts, in particular the slot and extension mechanism of our plug-in framework in .NET. Then Section 3 will discuss the different integration models between host and plug-ins and will discuss reliability and versioning issues. Section 4 will discuss the realization and Section 5 will conclude with a summary and an outlook to future research directions.

2 .NET Plug-in Architecture

In this section the basic concepts of our plug-in architecture for the .NET platform is reviewed. In particular, the slot and extension mechanism is shown. For a detailed elaboration the interested reader is referred to [Wo06].

The .NET plug-in platform shows much resemblance to Eclipse [Bg03]. So it supports an easy, file copy-based deployment and discovery mechanism, it allows lazy loading and static integration of plug-ins, and adopts an extension specification approach. However, it exploits and relies on .NET specific features and technology [Mo04]. In particular, whereas Eclipse describes extension points and extensions with dedicated XML configuration files, our architecture relies on .NET concepts such as custom attributes and metadata to specify relevant information directly in the source code of an application.

In our architecture a plug-in is a deployable .NET assembly which has explicit specifications of its slots and extensions. Slot specifications define how other plug-ins are intended to extend the functionality of this plug-in, whereas extension specifications define how this plug-in makes contributions to slots of others. Therefore, slot and extension specifications have to match. In essence, slots declare the types of information a plug-in expects and the extensions fill this information slots accordingly. In its simplest form, a plug-in specification is a structured list of name/value-pairs where the slot specifies the required names and value ranges and the extension specification defines appropriate values for the extension at hand. The component defining the slot is called the *extension host* and the component implementing the extension is called the *extension contributor*.

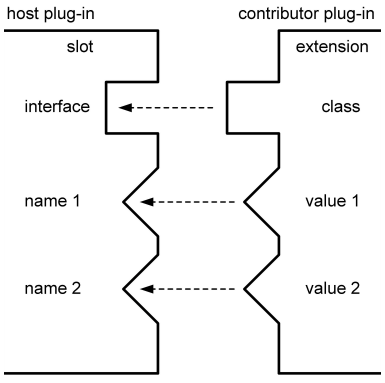

Figure 1: Slot and extension in host and contributor plug-in

Usually, plug-in extensions will occur on the level of run-time behaviour, i.e., plug-in host and contributor will communicate based on a defined protocol in order to accomplish a particular task. The collaboration between the host and its contributor is defined in the form of required and provided interfaces. The host will define the required interface and the extension contributor has to provide an implementation for it. Figure 1 depicts the structure of slot and extension specifications in host and contributor plug-ins. The interface in the host and the implementation class in the contributor specify the agreed collaboration protocol. Additional name/value pairs define other properties that the host requires to make use of the extension.

Specification of slots and extensions is based on .NET custom attributes [Mi01]. Custom attributes are pieces of meta-information that can be attached to language constructs such as classes, interfaces, methods or fields in the source code of an application. At run-time the attributes attached to a language construct can be retrieved using reflection. In addition to pre-defined attributes programmers can declare custom attributes by implementing attribute classes with arbitrary properties whose values can be set when the attribute is attached to a language construct.

The plug-in framework defines a set of custom attributes which are used for specifying slots and extensions. So a *Slot* attribute is used for tagging program elements which belong to a slot. The *Extension* attribute is used to tag the program element belonging to the extension. Additionally, slot specifications will define their own custom attributes to be used by their extensions.

Let us clarify the approach by a simple example. The example is taken from the workbench component of our prototype rich client platform. The workbench plug-in is the application's frame window and allows for an arbitrary number of actions to be plugged in. An action represents a command that can be triggered by the user. The following code snippet shows the specification of a simple *Action* slot, which allows extensions to contribute actions called from menus and toolbars. The interface *IAction* is specified to belong to the slot by the *Slot* attribute. The slot *Workbench.Action* is identified by a unique name. In the *RequiredAttribute* it is specified that an extension has to provide a *MenuItem* attribute. In the sequel this *MenuItem* attribute is defined by class *MenuItemAttribute*. It is intended to be used for specification of caption and icon of an action.

```
[Slot(Name="Workbench.Action")]
[RequiredAttribute("MenuItem")]
public interface IAction {
   void Do(object sender, EventArgs e); }
[Slot(Name="Workbench.Action")]
public class MenuItemAttribute : Attribute { ... }
```

An extension implementation of slot *Workbench.Action* now has to implement the *IAction* interface and provide the *MenuItem* attribute. The following code snippet shows a sample extension that contributes to slot *Workbench.Action.* The *Extension* attribute identifies the slot by its unique name. It uses the *MenuItem* attribute to define caption and icon to be used for this action in user interface widgets. Finally, the class *HelloAction* provides an implementation for the required slot interface *IAction*.

```
[Extension(Slot="Workbench.Action",Name="HelloAction")]
[MenuItem(Caption="Hello",Image="hello.ico")]
public class HelloAction : IAction {
  public void Do(object sender, EventArgs e) {
    MessageBox.Show("Hello world"); } }
```

3 Integration Models

In Section 1 we have discussed the issue of reliability and versioning for plug-in systems. In this section we incrementally develop four different plug-in integration models and discuss problems and advantages of each of them [Mi05]. In Section 4 we will then show how those are realized with .NET specific concepts and how they are supported in our .NET plug-in framework.

3.1 Tightly Coupled

The first one is to start with a tightly coupled model. In .NET, tightly coupled means that host and plug-in are in the same application domain as shown in figure 2. An application domain can be perceived as a logical process with memory isolation that is built on top of an operating system process [Lo03]. If both, host and plug-in assembly, share a single application domain, then there is no kind of isolation possible.

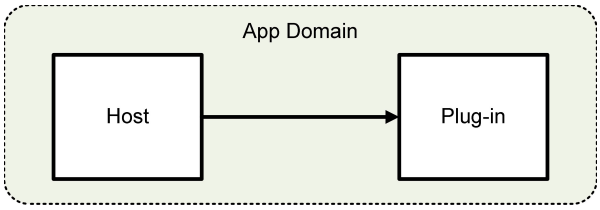

Figure 2: Tightly coupled model

There are four problems with this tight coupling:

Problem #1: A tightly coupled plug-in cannot be unloaded. As the .NET Common Language Runtime (CLR) does not have the ability to unload individual assemblies, a plug-in once loaded cannot be unloaded.

Problem #2: Same CLR and framework for host and plug-in. If host and plug-in reside in the same application domain, they have to agree to use the same version of the CLR and they have to use the same version of all the frameworks.

Problem #3: Versioning - If host is updated, plug-ins must be recompiled. When the host application is updated with a new version and if the plug-in was using types of the host, the plug-in must be recompiled with the new types[1]. This impedes independent versioning of host and plug-ins.

Problem #4: Unsafe plug-in can cause system crashes. As host and plug-in operate in a single CLR, a faulty plug-in doing unmanaged calls can pull down the whole application.

3.2 Isolation Boundary

The obvious starting place, to solve those four problems, is to put an isolation boundary between the host and the plug-in. In particular to put them in a separate application domain is an obvious approach in the .NET framework.

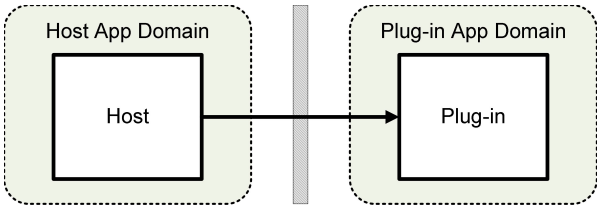

Figure 3: Model with application domain isolation boundary

Separate application domains solve problem #1, problems #2 to #4 remain. Now plug-ins can be unloaded because the CLR can unload the plug-in application domain. Unfortunately the isolation boundary brings in two new problems that we did not have before.

Problem #5: Application domain boundary requires remoting. Application domains in .NET are separate memory spaces and communication between program elements in separate application domains have to be done using .NET remoting. That means, in order to cross the boundary, an object has to be either serializable or it has to be a marshal-by-reference object.

Problem #6: Performance penalty. That other new problem is, that remoting brings in a performance penalty.

3.3 Stable Contract

If we want to be version resilient, i.e. we want to make sure that we could actually update the host application independently, a way is the introduction of a stable contract as boundary between host and plug-in.

---

[1] This is not an issue with assemblies that are not strong-named [Ri02, 72f]. Assemblies without strong-names are considered by the CLR as presumably type compatible. As long as public interfaces are not broken, there will not be type mismatch errors.

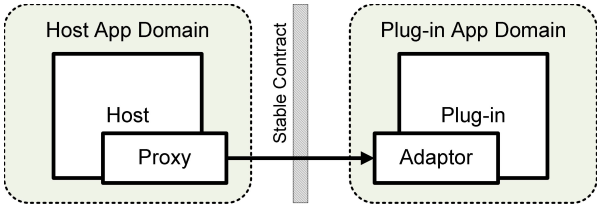

Figure 4: Stable contract model

The contract actually is supposed not to change. Host and plug-in rely on this stable contract and will be developed against this stable contract. This results in backward and forward compatibility. Host and plug-in now can evolve independently and will stay compatible. To make the stable contract work, a proxy is used on the host and an adaptor object is used at the plug-in side.

The version resilient model solves problem #3, i.e. the versioning problem, but adds two new problems:

Problem #7: Stable contract challenge. It is certainly a challenge to design a contract between a host and the plug-ins which should not change in the future.

Problem #8: Implementation of proxy and adaptor. It represents an additional effort to implement the proxy on the host side and that adaptor on the plug-in side.

3.4 Cross-Process Boundary

To provide even better isolation between host and plug-in, one can use independent processes instead of an application domain.

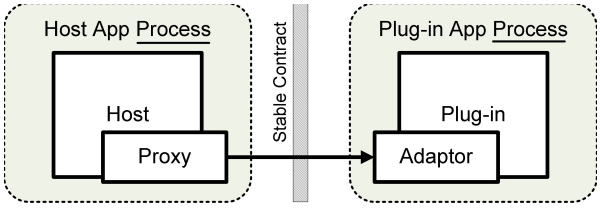

Figure 5: Model with cross-process isolation boundary

That model solves the remaining problems #2 and #4. Now each process can have its own CLR and its own .NET Framework. They are completely independent and using a stable contract they are version resilient.

Problem #6 (worse): Performance penalty even higher.

Problem #9: Cannot use process-local resources. The plug-in cannot use any process-local resources across that boundary.

Problem #10: Thread synchronization issues. Since it is no longer the same thread on both sides we get a threading and re-entrancy problem.

Problem #11: Process life-cycle management. The framework has to host the plug-in processes and it has to manage their life-cycle, e.g. it has to notice when a process goes away.

4 Realization

In this section we present how the integration models from Section 3 are implemented in our prototype plug-in framework. Our plug-in framework provides solutions for all seven basic mechanism presented in Section 1.1. In this section we focus on those mechanisms that are affected by the integration models to provide reliability and versioning.

4.1 Contract Specification

Section 2 presented how we use slots and extensions to specify explicitly how a component should be extended and how other plug-in components make their contributions. The following code snippet revisits the *Workbench.Action* slot definition and the corresponding *HelloAction* extension from Section 2 and adds some additional properties. An *AssemblyVersion* attribute is added to enable version control on the contract. And in order to support the isolated integration models, either application domain or cross-process isolation, the *Slot* attribute gets additional properties to specify a proxy class and a proxy assembly.

```
[assembly: AssemblyVersion("1.0.*")]
[Slot(Name="Workbench.Action",Proxy="ActionProxy",Assembly="Workbench.Proxy")]
[RequiredProperty("MenuItem")]
public interface IAction {
  void Do(object sender, EventArgs e); }

[assembly: AssemblyVersion("1.0.*")]
[Extension(Slot="Workbench.Action",Name="HelloAction",AutoLoad=false,
  Dependencies = new string[] { "Workbench" })]
[MenuItem(Caption="Hello",Image="hello.ico")]
public class HelloAction : IAction {
  public void Do(object sender, EventArgs e) {
    MessageBox.Show("Hello world"); } }
```

4.2 Deployment

We utilize .NET assemblies as unit of deployment. To enable version control we added version information through the *AssemblyVersion* attribute and strong name identification [Ri02, 72f]. The make the contract stable, the workbench plug-in providing the slot *Workbench.Action* is built into two separate assemblies. A stable contract never versions and does not change. Thus the slot definition for the workbench must be in its own separate assembly *Workbench.Contracts.v1.dll*, apart from the workbench plug-in itself which resides in the assembly *Workbench.dll*. Plug-in assemblies reference that contract assembly and in order to not break the types referenced by the plug-in it is crucial to not recompile the contract assembly after it has shipped. To not break the contract, any sub-

sequent change or addition to the slot definition results in a completely new contract assembly *Workbench.Contracts.v2.dll*. With the contract put in its separate assembly, the implementation of the workbench component can be versioned without breaking any of its plug-ins. Figure 6 shows the build schema to make the contract stable.

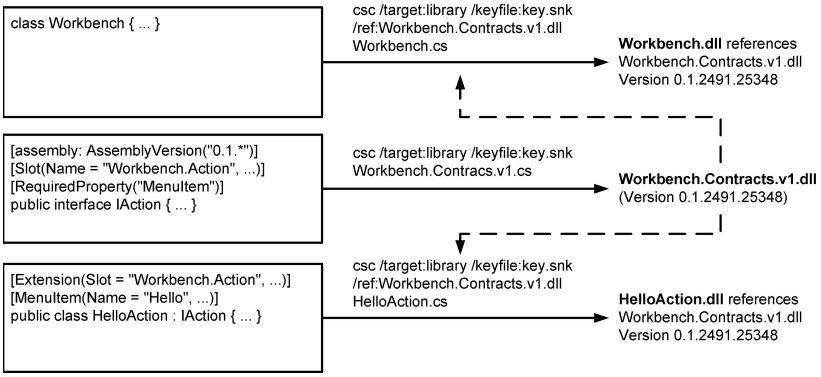

Figure 6: Build schema for stable contract Workbench.Contracts.v1.dll

4.3 Qualification

Qualification means to make sure that a plug-in is appropriate for using with a host, or to be more specific, whether an extension is appropriate for a specific slot. A slot requires an interface to be implemented and a number of values to be set for certain custom properties. Before a host can activate an extension, the framework checks whether that extensions fulfills all those requirements.

Qualification is also about negotiating constraints. The host may put certain restrictions on the integration model used for a particular slot. Or a plug-in might have requirements, e.g. a plug-in might want to run in a separate process or cannot run in a separate process. If a host has reliability concerns for a slot, it might intend to run plug-ins in a separate process only. The framework negotiates and get both sides to agree.

4.4 Activation

The framework activates a plug-in in an environment that satisfies the constraints on both the host and the plug-in. Activation is where the integration models from Section 3 really come into play. The plug-in uses a custom attribute *IntegrationMode* to specify its integration mode requirements. Due to its impact on how an assembly is loaded by the CLR, the integration mode must be specified for a whole plug-in assembly and not per individual extension. The framework supports three levels of isolation for integration:

(a) HostAppDomain. The plug-in wants to be activated in the application domain of its host. Therefore it will be tightly coupled to that host, which means unrestrained performance but the ability to be unloaded is tied to the host. If the host can be unloaded, so can the plug-in, but only in association with the host.

(b) AppDomain. The plug-in wants to be activated in a separate application domain. Communication between host and plug-in means remoting which brings a performance penalty. The plug-in can be individually unloaded. Optionally the plug-in can specify a friendly name for the application domain. By using the same friendly name, plug-ins can consolidate into a shared application domain, which enhances performance within this domain but requires those plug-ins to be unloaded in association.

(c) Process. The plug-in wants to be activated in a separate operating system process. Communication between host and plug-in means remoting cross processes which brings an even higher performance penalty compared to application domain isolation. The separate process can be individually stopped and destroyed.

If a plug-in omits to specify an integration mode, the default mode *HostAppDomain* is used. Specifying the integration mode is as simple as setting the custom attribute *IntegrationMode* as shown below. To change the integration mode later on is as easy as changing one attribute value. No further coding work in the plug-in. Everything is done by the framework. So the implementation of a plug-in is not affected by the integration mode it is going to use.

```
[assembly: IsolationMode(AppDomain,FriendlyName="IsolatedHelloWorld")]
[Extension(Slot="Workbench.Action",Name="HelloAction", ...)]
public class HelloAction : IAction { ... }
```

However there is some additional coding effort in the host, if it wants to support application domain or process isolation. The host needs to implement proxies for each of its slots. As shown in the code snippet in Section 2 on page 9 the host needs to specify a proxy class and a proxy assembly. The proxy class must inherit from *MarshalByRefObject* to enable remoting. Implementation of that proxy class is a routine task, since it is simply a wrapper around the interface definition of the slot. We plan to take that implementation burden of the plug-in contributor by providing on-the-fly generated proxies in future generations of the framework.

4.5 Deactivation

If a plug-in or a combination of plug-ins is used temporarily for a specific task we might want to get rid of it. We want to deactivate the plug-in, i.e., remove it from the user interface. In many cases we even want to unload the implementation and regain resources. The plug-in framework can do this. And the important thing is, that it can do this without restarting the host-application. The framework is monitoring plug-ins for removals. When it detects that a plug-in has been removed from the repository the framework notifies the host who can disintegrate UI elements before the framework destroys the extension instances. Depending on whether the plug-in had been isolated in a separate application domain or process, the plug-in assembly is unloaded. Either immediately if the plug-in had its own application domain, or deferred as soon as all other plug-ins within the same domain have been deactivated.

Deactivation of plug-ins at run-time is the most important requirement for implementing a hot-update capability. Hot update means to replace a running plug-in with a newer version without restarting the application. With our plug-in framework, if a new version of

a currently active plug-in is installed to the repository by overwriting the previous version, the framework will automatically unload the old version and activate the new version. The rest of the application can keep running.

5 Summary and Outlook

In this paper we presented the basic problem areas when dealing with plug-ins and we presented integration models that address issues that emerge when plug-in systems are applied in the enterprise domain. Those integration models provide the groundwork for security, reliability and versioning. The realization of the integration models poses implementation challenges that need to be concealed from a plug-in developer. Our plug-in framework shows that .NET concepts can be applied to accomplish those challenges inside the framework itself.

In this paper we discussed objectives, architecture, basic concepts, and implementation issues of a new plug-in framework for the .NET platform. The plug-in framework is intended to serve as a foundation for application software in the enterprise domain. A major goal is that applications should become inherently extensible and customizable. The plug-in framework shows much resemblance to Eclipse but uses .NET specific features for plug-in contract specification, deployment, discovery, qualification, integration, activation and deactivation.

In the enterprise application domain, reliability and security requirements play a dominant role. Additionally, the independent versioning of host and plug-ins has been identified as an indispensable demand from our industrial partner. This paper has described different integration models between host and plug-ins. We have shown how the different integration models solve the reliability and versioning problem and have shown how they are supported by our current plug-in framework.

However, we have not tackled the problem of security so far, but this will be subject of a continuing research step. Two main directions will be pursued. First we will work out how the .NET security model with code access security and role-based security match with the plug-in approach, in general, and the different integration models, in particular. Second, we will investigate how behavioral contract specifications [Sc04][Zs05][Bl06] and protocol verification approaches can help that a host application can protect itself from faulty and possibly malicious plug-ins.

6 References

[Bg03] Beck, K., Gamma, E.: Contributing to Eclipse. Addison-Wesley, 2003.

[Bl06] Bläser, L.: A Component Language for Structured Parallel Programming. Lecture Notes in Computer Science, Vol. 4228, Proceedings of 7th Joint Modular Languages Conference, JMLC 2006 Oxford, UK, September 13-15, 2006.

[Ec03] Eclipse Platform Technical Overview. Object Technology International, Inc., http://www.eclipse.org, February 2003.

[Lo03] Löwy, J.: Programming .NET Components. O'Reilly, 2003.

[Mi01] Microsoft: Microsoft C# Language Specifications. Microsoft Press, Redmond (2001).

[Mi05] Miller, J., Quinn, T.: CLR: Designing Managed AddIns for Reliability, Security and Versioning. Session FUN309, Microsoft Professional Developers Conference, September 5, 2005.

[Mo04] Mössenböck, H., Beer W., Birngruber, D., Wöß, A.: .NET Application Development. Pearson Addison Wesley, 2004.

[Ri02] Richter, J.: Applied Microsoft .NET Framework Programming. Microsoft Press, 2002.

[Sa00] Shaver, M., Ang, M.: Inside the Lizard: A Look at the Mozilla Technology and Architecture. http://www.mozilla.org, 2000.

[Sc04] Schmidt, H. W. et al.: Predictable Component Architectures Using Dependent Finite State Machines. In: Wirsing, M.: Knapp, A., Balsamo, S. (eds.), Radical Innovations for Software and Systems Engineering in the Future. 9th International Workshop, RISSF 2002. Springer-Verlag, 2004.

[Wo06] Wolfinger, R., Dhungana, D., Prähofer, H., Mössenböck, H.: A Component Plug-in Architecture for the .NET Platform. Lecture Notes in Computer Science, Vol. 4228, Proceedings of 7th Joint Modular Languages Conference, JMLC 2006 Oxford, UK, September 13-15, 2006.

[Zs05] Zulkernine, M., Seviora, R.: Towards automatic monitoring of component-based software systems. J. Syst. Softw. 74 (1), Jan. 2005, pp. 15-24.