Plux.NET - A Dynamic Plug-in Platform for Desktop and Web Applications in .NET

Markus Jahn[1], Markus Löberbauer[1], Reinhard Wolfinger[2], Hanspeter Mössenböck[2]

Christian Doppler Laboratory for Automated Software Engineering[1]
Institute for System Software[2]
Johannes Kepler University
Altenberger Straße 69
A-4040 Linz
{markus.jahn | markus.loeberbauer | reinhard.wolfinger | hanspeter.moessenboeck}@jku.at

Abstract: Plug-in frameworks support the development of component-based software that is extensible and can be customized to the needs of specific users. However, most plug-in frameworks target desktop applications and do not support web applications that can be extended by end users. In contrast to that, our plug-in framework Plux supports desktop as well as web applications. Plux tailors applications to the needs of every user, by assembling it from a user-specific component set. Furthermore, Plux supports end-user extensions, by integrating components provided by the end user, even into web applications. Plux supports distributed web applications, by integrating components on the client machines into the web application. Plux allows application developers to restrict who is allowed to extend an application, at which points the application can be extended by a specific third party, and which operations such extensions are allowed to perform. And finally, Plux allows developers to retrofit security around unsecured components by specifying security constraints declaratively.

1 Introduction

Although modern software systems tend to become more and more powerful and feature-rich they are still often felt to be incomplete. It will hardly ever be possible to hit all user requirements out of the box, regardless of how big and complex an application is. One solution to this problem are plug-in frameworks that allow developers to build a thin layer of basic functionality that can be extended by plug-in components and thus tailored to the needs of specific users.

Most plug-in frameworks target desktop applications, but are typically unsuitable for building extensible web applications. For us, a web application is a program that is concurrently used by multiple persons, over a network using a web browser. In domains where customer requirements vary greatly (e.g., in business software) a web application should be extensible by end users to meet their specific needs.

Making a web application extensible must go beyond componentization. Every user should be able to extend the application in his own way, i.e. he should be able to add custom extensions without changing the application for other users. We are aware that extensions provided by end users raise security concerns. Therefore Plux implements a security mechanism based on signed assemblies and .NET code access security. Depending on the identity of the manufacturer, Plux decides if a component can be loaded and in which parts of the application it can be used. Components can also be partially trusted, in that case they can be executed in a sandbox with limited rights.

Distribution of components across several computers is another issue, because installing components only on the server does not cover all extensibility scenarios. For example, if a component needs to access client-side hardware, such as a barcode scanner, the component must run on the client and not on the server. Such client-side components should be capable of being integrated into the web application as well.

Over the past few years we have developed the plug-in framework Plux. Originally, Plux focused on dynamically reconfigurable desktop applications. Lately we extended Plux for web applications and addressed the problems of extensibility and distribution. A Plux web application can be extended by custom plug-ins both on the server-side and on the client-side. Plug-ins which are distributed across multiple computers can still be integrated into a single seamless application.

Our research was conducted in cooperation with BMD Systemhaus GmbH. BMD is a medium-sized company offering a comprehensive suite of enterprise applications, such as customer relationship management, accounting, production planning and control. Because BMD's target market is fairly diversified, ranging from small tax counsellors to large corporations, customization and extensibility are essential parts of BMD's business strategy. As BMD offers both a desktop and a web version of their software, they want to use Plux for both versions and reuse components where possible.

This paper is organized as follows: Section 2 describes the concepts of the Plux framework. Section 3 describes the architecture of the Plux composition infrastructure and the automatic composition process. Section 4 discusses related work. It describes to what extent current desktop plug-in frameworks can be used to build extensible web applications and how non-plug-in-based web development platforms address extensibility. Section 5 finishes with a summary and an outlook to future work.

2 Concepts of Plux

The Plux framework supports the dynamic composition of applications using a plug-and-play approach [1]. It facilitates extensible and customizable applications that can be reconfigured without restarting them. Reconfiguring applications can be done in a plug-and-play manner and does not require any programming. If a user wants to add a feature, he just drops a plug-in (i.e., a DLL file) into a directory. Plux discovers the plug-in on-the-fly and integrates it into the application without requiring a restart. Similarly, if the user wants to remove a feature, he removes the corresponding plug-in from the directory.

Together with our industrial partner BMD, we applied Plux to their customer relationship management (CRM) product [2]. By allowing dynamic addition and removal of CRM features, we support a set of new usage scenarios, such as on-the-fly product customization during sales conversations or incremental feature addition for step-by-step user trainings [3].

The main characteristics of Plux are: the *composer*, the *composition events*, the *composition state*, and the *replaceable component discovery mechanism*. These characteristics distinguish Plux from other plug-in systems [4], such as OSGi [5], Eclipse [6], and NetBeans [7], and allow Plux to replace programmatic composition by automatic composition. *Programmatic composition* means that components query a service registry and integrate other components programmatically. *Automatic composition* means that the components declare their requirements and provisions using metadata; the *composer* in Plux uses these metadata to match requirements and provisions and to connect matching components automatically. During composition, Plux sends *composition events* to which the affected components can react. Plux also maintains the *current composition state*, i.e. it stores which components use which other components. As components can retrieve the global composition state, they do not need to store references to the components they use. *Discovery* is the process of detecting new components and extracting their metadata. Unlike in other plug-in systems, the discovery mechanism is not an integral part of Plux, but is a plug-in itself. This makes the mechanism replaceable. The following subsections cover those characteristics in more detail.

2.1 Metadata

Plux uses the metaphor of extensions that have slots and plugs (Fig. 1). An *extension* is a component that provides services to other extensions and uses services provided by other extensions. If an extension wants to use a service of some other extension it declares a *slot*. Such an extension is called a *host*. If an extension wants to provide its service to other extensions it declares a *plug*. Such an extension is called a *contributor*.

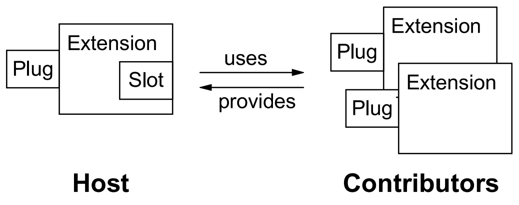


Figure 1: Extensions with slots and plugs

Slots and plugs are identified by names. A plug matches a slot if their names match. If so, Plux will try to connect the plug to the slot. A slot represents an interface, which has to be implemented by a matching plug. The interface is specified in a *slot definition*. A slot definition has a unique name as well as optional parameters that are provided by the contributors and retrieved by the hosts. The names of slots and plugs refer to the respective slot definitions.

The means to provide metadata is customizable in Plux. The default mechanism extracts metadata from .NET attributes in assembly files. Attributes are pieces of information that can be attached to .NET constructs, such as classes, interfaces, methods, or fields. At run time, the attributes can be retrieved using reflection [8].

Plux has the following custom attributes: The *SlotDefinition* attribute to tag an interface as a slot definition, the *Extension* attribute to tag classes that implement components, the *Slot* attribute to specify requirements for optional contributors in hosts, the *Plug* attribute to specify provisions in contributors, the *ParamDefinition* attribute to declare required parameters in slot definitions, and the *Param* attribute to specify provided parameter values in contributors. Although the default mechanism for providing metadata (namely by .NET attributes) limits parameter values to compile-time constants, Plux in general can use arbitrary objects as parameter values.

Let us look at an example now. Assume that a host wants to print log messages as errors or warnings. The loggers should be implemented as contributors that plug into the host. Every logger must use a parameter to specify whether it prints errors or warnings. First, we have to define the slot into which the logger can plug (Fig. 2).

```
public enum LoggerKind { Warning, Error }

[SlotDefinition("Logger")]
[ParamDefinition("Kind", typeof(LoggerKind))]
public interface ILogger {
  void Print(string msg);
}
```

Figure 2: Definition of the *Logger* slot

Next, we write logger contributors. Fig. 3 shows the logger for errors. The logger for warnings is implemented similarly (not shown). Since *ErrorLogger* has a *Logger* plug it has to implement the interface *ILogger* specified in the slot definition of *Logger*. It also has to provide a value for the parameter *Kind* specified in the slot definition.

```
[Extension]
[Plug("Logger")]
[Param("Kind", LoggerKind.Error)]
public class ErrorLogger : ILogger {
  public void Print(string msg) {
    Console.WriteLine(msg);
  }
}
```

Figure 3: *ErrorLogger* as a contributor for the *Logger* slot

Finally, we implement the application that uses the loggers (Fig. 4). In order to be able to use loggers it has a *Logger* slot. It also has an *Application* plug that fits into the *Application* slot of the Plux core. At startup, Plux creates an instance of *HostApp* and connects it to the core. The full implementation of *HostApp* is shown in Section 2.4.

```
[Extension]
[Plug("Application")]
[Slot("Logger")]
public class HostApp : IApplication {
  public HostApp(Extension e) { ... }
  void Work() { ... }
}
```

Figure 4: Application host with a *Logger* slot

2.2 Discovery

In order to match requirements and provisions, Plux uses the metadata of the extensions. Extensions are deployed as plug-ins, i.e. DLL assembly files. A plug-in can contain several functionally related extensions that should be jointly installed. The discoverer is the part of Plux which discovers plug-ins and provides the metadata for the extensions in the plug-in. Plux supports dynamic discovery, i.e. plug-ins can be added and removed without restarting the application. The default discoverer reads the metadata from attributes stored in the plug-in assemblies. As the discoverer is an extension itself, one can write custom discoverers, e.g., to retrieve metadata from a database or from a configuration file.

2.3 Composition

Composition is the mediating process which matches the requirements of hosts with the provisions of contributors. In Plux, this is done by the composer. The composer assembles programs from the extensions provided by the discoverer. Thereby it connects the slots of hosts with the plugs of contributors.

When the discoverer provides a new extension, the composer integrates it into the program on-the-fly. Similarly, if an extension is removed from the plug-in repository, the composer removes it from the program.

Integrating an extension means, that the composer instantiates it and connects its plugs with the matching slots of extensions in the program. If a plug is connected to a slot, we call this relationship *plugged*. Removing an extension means that the composer unplugs the instances of this extension from the slots where they are plugged, i.e. it removes the plugged relationship for the corresponding slots and plugs.

Slots can declare whether they want an instance of their own or a shared instance of a contributor. The composer connects a new instance to slots that want their own instance and the same (shared) instance to slots that want the shared instance.

2.4 Composition State

In Plux, all connections between components are established by the composer. Therefore the composer has full knowledge about the instantiated extensions, their slots and plugs as well as about their connections. This is called the *composition state*. If a host wants to use its plugged contributors, it can simply retrieve them from the composition state. For every instantiated extension, the composition state holds the *meta-object* of the extension, the meta-objects of its slots and plugs as well as a reference to the corresponding *extension object* (Fig. 5). For every slot, the composition state also indicates which plugs are connected to this slot.

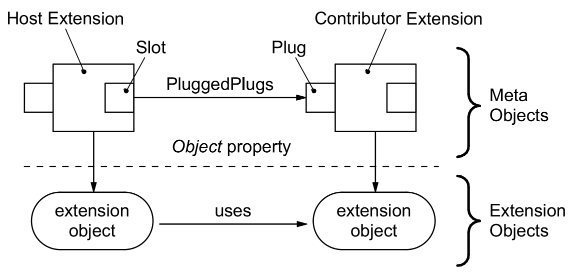


Figure 5: Meta-objects for instantiated extensions in the composition state

Fig. 6 describes the host of Fig. 4 in more detail showing how meta-objects can be used by a program. When the composer creates an extension it passes the extension's meta-object to the constructor. In Fig. 6, the constructor retrieves the meta-object of the slot *Logger* and starts a new thread. In the *Run* method, the host does its work and uses the connected loggers to print a message. It retrieves the loggers using the *PluggedPlugs* property of the logger slot. For each logger, it checks the logger kind using the parameter *Kind*. Finally, it retrieves the extension objects for loggers of the desired kind and prints the message.

```
[Extension]
[Plug("Application")]
[Slot("Logger")]
public class HostApp : IApplication {
  Slot s; // logger slot
  public HostApp(Extension e) {
    s = e.Slots["Logger"];
    new Thread(Run).Start();
  }
  void Run() {
    while(true) {
      string msg; LoggerKind kind;
      Work(out msg, out kind);
      foreach(Plug p in s.PluggedPlugs) {
        if((LoggerKind) p.Params["Kind"]
            == kind) {
          Extension e = p.Extension;
          ILogger logger = (ILogger)e.Object;
          logger.Print(msg);
        }
      }
      Thread.Sleep(2000);
    }
  }
  void Work(out string msg,
      out LoggerKind kind) {
    /* not shown */
  }
}
```

Figure 6: Application host using logger contributors

2.5 Composition Events

In addition to accessing the composition state, a host can listen to composition events. This is appropriate for hosts that want to react to added or removed contributors immediately, e.g., in order to show them in the user interface. Fig. 7 shows a modified version of our host from Fig. 6. It uses the

Slot attribute to register event handler methods for the *Plugged* and *Unplugged* events. In this example, the event handlers just print out which logger was plugged or unplugged.

```
[Extension]
[Plug("Application")]
[Slot("Logger",
  OnPlugged="Plugged",
  OnUnplugged="Unplugged")]
public class HostApp : IApplication {
  ...
  void Plugged(CompositionEventArgs args) {
    Extension e = args.Plug.Extension;
    ILogger logger = (ILogger) e.Object;
    logger.Print("plugged: " + e.Name);
  }
  void Unplugged(CompositionEventArgs args) {
    ...
    logger.Print("unplugged: " + e.Name);
  }
  void Run() { ... }
  void Work(...) { ... }
}
```

Figure 7: Modified application reporting connected contributors

This completes the example. We compile the slot definition interface *ILogger* to a DLL file, the so-called *contract* assembly. Contracts and plug-ins should be separate DLL files, because bundling slot definitions with extensions would constrain customization. We could not use a slot definition in a program without also including the extensions that come with it. If we compile the classes *ErrorLogger* and *HostApp* to plug-in DLL files and drop them into the plug-in repository of Plux everything will fall into place. The Plux infrastructure will discover the extension *HostApp* and plug it into the *Application* slot of Plux. It will also discover the extension *ErrorLogger* and plug it into the *Logger* slot of *HostApp* (Fig. 8).

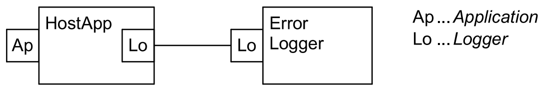


Figure 8: Composed application with host and logger contributor

2.6 Lazy Activation

In order to minimize startup time and memory usage, Plux supports lazy activation of extensions. This means that contributors are only instantiated on demand, i.e. when the host accesses the extension's *Object* property. For the example in Fig. 6 this has the effect that loggers which do not match the desired kind are not instantiated; only their meta-objects exist.

If a contributor is no longer needed and the host wants to release the resources used by it, the host can deactivate the contributor. To deactivate means to free the extension object, so that only its meta-object remains. On the next access to the *Object* property, the contributor is automatically reactivated.

Hosts can listen to the composition events *Activated* and *Deactivated* if they want to distinguish between activated and deactivated contributors. Typically such a host cooperates with another host. The first host handles only activated contributors while the other one controls which contributors get activated. For example, a window host might show a child window for every activated contributor, while a menu host might allow the user to activate and deactivate contributors causing child windows to be opened or closed.

2.7 Programmatic Composition

The mechanism described in the previous sections, where the composer makes connections and extensions retrieve connections, is called automatic composition. In addition to that, hosts can assemble contributors using programmatic composition, i.e. the host can control how the composer assembles the program. For example, the host can use API calls to integrate specific contributors, a script interpreter can assemble a program from a script, or a serializer can restore a previously saved program.

2.8 Composition Restrictions

The developer of an extension can manage composition restrictions for it, i.e. constrain to which other extensions it is allowed to contribute, and constrain which other extensions are allowed to be hosted in its slots. Composition restrictions are specified with attributes: the *AllowPlugTo* attribute specifies where the extension can be plugged, and the *AllowPlugFrom* attribute specifies which extension can be plugged into the slots of the extension. Extensions are identified using the digital signatures specified in the certificate files. Figure 9 shows an example for composition restrictions. They ensure that the host application plugs only to hosts from the manufacturer of Plux, and that only contributors from certified partners of the application are plugged.

```
[Extension]
[Plug("Application")]
[AllowPlugTo("Plux.cer")]
[Slot("Logger",  ...)]
[AllowPlugFrom("CertifiedPartners.cer")]
public class HostApp : IApplication { ... }
```

Figure 9: Extension with in-component restrictions

2.9 Retrofitted Security

Plux comes with a security library that can be used to retrofit security. This library contains three predefined security devices: the composition interceptor, the call interceptor, and the sandbox. For scenarios which are not covered by the predefined devices, developers can program custom devices.

2.9.1 Composition Interceptor

The composition interceptor can be used to prevent the integration of untrusted plug-ins. It can block discovery, instantiation, and connection of components. Thereby, a plug-in can be identified by different evidences, e.g., the location of a plug-in file, the signature of a plug-in, or the origin of a plug-in.

Security devices are configured using restrictions. A restriction specifies a device name and multiple parameteres which comprise a name and value. Figure 10 shows two example restrictions for composition interceptors: The first restriction blocks connections from a contributor *DataStore* in the plug-in *Data.dll* to extensions from plug-ins that are located outside of the trusted folder. The second restriction goes even further and blocks all instantiations from plug-ins outside the trusted folder.

```
restriction(CompositionInterceptor)
      action = "blockConnection"
      host = "not in (plugins/trusted/**)"
      contributor = "plugins/Data.dll/DataStore" .
restriction(CompositionInterceptor)
      action = "blockInstantiation"
      extension = "not in (plugins/trusted/**)"
```

Figure 10: Example configuration file for composition interceptors

2.9.2 Call Interceptor

The call interceptor can be used to control the data flow between components. It can block calls to components, blank out return values and sanitize arguments, and log and report component calls.

Figure 11 shows an example restriction for the call interceptor. The restriction blocks calls to write methods on connections which use the slot definition Data. The restriction applies to plug-ins which are located outside the trusted folder. As the call interceptor needs to distinguish between read-only and write methods, the methods must be annotated in the interface of the slot definition using read and write attributes (cf. Figure 12). If the slot definition does not contain these annotations, the developer must implement a custom call interceptor.

```
restriction(CallInterceptor)
     action = "blockComponentCall"
     slot = "Data"
     attribute = "Write"
     host = "not in (plugins/trusted/**)"
     name = "WriteProtector" .
```

Figure 11: Example configuration file for a generated call interceptor

```
SlotDefinition("Data")]
interface IData {
    [Read] object Get(...);
    [Write] void Put(Object ...);
}
```

Figure 12: Slot definition with classified read and write methods

2.9.3 Sandbox

The sandbox can be used to restrict security-relevant functionality, e.g., networking, file access, and introspection. It blocks calls to the system libraries. Figure 13 shows an example restriction for a sandbox. The restriction requires a sandbox in which extensions cannot access the network. The restriction applies to hosts from the untrusted folder to which contributors from the trusted folder are connected.

```
restriction(Sandbox)
     permissionSet = "NetworkingDisabled"
     host = "plugins/untrusted/**"
     contributor = "plugins/trusted/**"
     target = "host" .
```

Figure 13: Example configuration file for a sandbox

2.10 Web Application Support

Web applications face similar problems as desktop applications: If they get big and feature-rich, they become hard to understand and difficult to maintain. They are hardly customizable and usually not extensible by end users. Furthermore, web applications cannot access the local hardware of client computers. In order to solve these problems we applied the plug-in approach also to web-based software. While the original version of Plux targeted single-user desktop applications, we enhanced it so that it can be used to build multi-user web applications.

Plux makes web applications extensible. Extensions can either be installed by the administrator or by the end user. Authorized users can install them directly on the server, while non-authorized users can install them on the client. Regardless of where an extension is installed, it is always seamlessly integrated into the web application.

Plux allows setting different user scopes. Extensions can be made available for *all users* of a web application, for a *group of users*, or for a *single user*. Thus every user can have an individual set of components, i.e., an individual composition state.

Depending on their type of integration, extensions are classified into three categories: a) *Server-side* extensions are installed and executed on the server. b) *Client-side* extensions are installed and executed on the client. c) *Sandbox* extensions are installed on the server, but executed in a sandbox on the client. Regardless of where the extensions are executed, Plux composes them into a coherent web application giving the user a seamless experience. The composition in Fig. 14 shows the different integration types. The server-side extensions *Su*, *E1*, *E2*, and *E3* constitute the base configuration that is available to all users. The server-side extension *E4* is user-specific and therefore available only for a single user. The client-side extensions *E5* and *E6* are user-specific extensions that are remotely plugged into the web application as well as the sandbox extensions *E7*, *E8*, and *E9*.

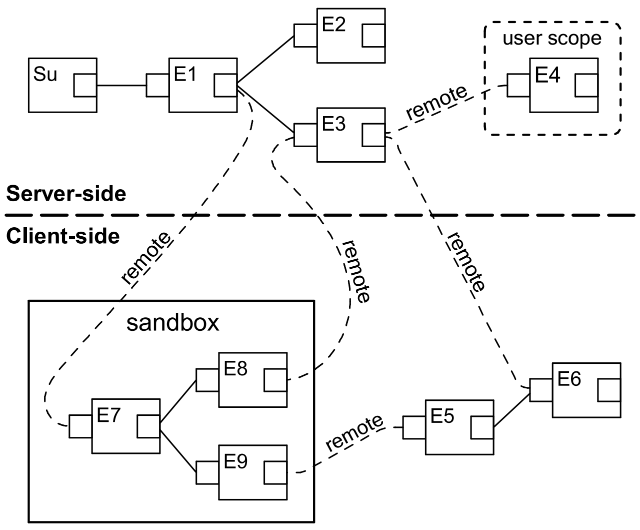


Figure 14: Composition with server-side, client-side and sandbox extensions

Server-side, client-side and sandbox extensions are implemented in exactly the same way. The different modes in which they are composed and executed are managed by Plux. Therefore, a server-side extension of one web application can be reused as a client-side extension in some other web application and vice versa. The only exception are client-side extensions that use local devices: these extensions have to run on the client on which this device is installed.

2.10.1 Server-side Extensions

The server-side extensions are installed and executed on the server. Because they are executed on the same server as the base components, there is no performance penalty caused by remote communication and they are available regardless from which computer the user connects. However, server-side extensions increase the work load on the server and may execute malicious code. For that reason, user-specific server-side extensions are executed in an individual user scope and users typically need to be authorized to install extensions on the server.

2.10.2 Client-side Extensions for Single Users

Client-side extensions are installed on the client and are remotely plugged into the web application. To plug remotely means that the host and the contributor are executed on different computers. Plux creates proxies on both sides on-the-fly. These proxies handle the communication between the host and the contributor transparently, i.e., Plux allows every extension to run remotely without any special coding effort.

Client-side extensions allow users to build components that integrate local hardware or software into the web application. Furthermore, since client-side extensions are installed on client computers, they open the web application also for extensions of users who are not authorized to install extensions on the server. The disadvantage of client-side extensions is that the remote execution of extensions causes some communication overhead.

2.10.3 Sandbox Extensions

Like client-side extensions, the sandbox extensions in this scenario are executed on the client, but unlike client-side extensions, they are installed on the server and downloaded to the client on demand. On the client, they are executed in a sandbox, e.g., in the Silverlight environment. This integration type is useful for building rich user interfaces for web applications. Because sandbox extensions are executed on the client, they reduce the work load on the server.

2.11 More Features

Other features of Plux that cannot be discussed at length here are *slot behaviors* that allow developers to specify how slots behave during the composition (e.g., one can limit the capacity of a slot to *n* contributors, or automatically remove a contributor from a slot when a new contributor is plugged), *component templates* to define generic extensions, that can be reused with different metadata in different parts of the program, as well as a *scripting API* that allows experienced users to override the operations of the composer. For a more extensive description of these features see [1, 9, 10].

3 Composition Infrastructure of Plux

The composition infrastructure builds programs from contracts and plug-ins. It discovers extensions from a plug-in repository and composes the program from them by connecting matching slots and plugs. The plug-in repository is typically a directory in the file system containing contract DLL files (with slot definitions) and plug-in DLL files (with extensions).

3.1 Architecture

Fig. 15 explains the subsystems of the composition infrastructure and how they interact. The *discoverer* ensures that at any time the type store contains the metadata of extensions and slot definitions from the plug-in repository. When the discoverer detects an addition to the repository, it extracts the metadata from the DLL file and adds them to the type store. Vice versa, when it detects a removal from the repository, it removes the corresponding metadata from the type store. The discoverer is implemented as an extension itself. Thus, it can be replaced with or extended by other discoverers.

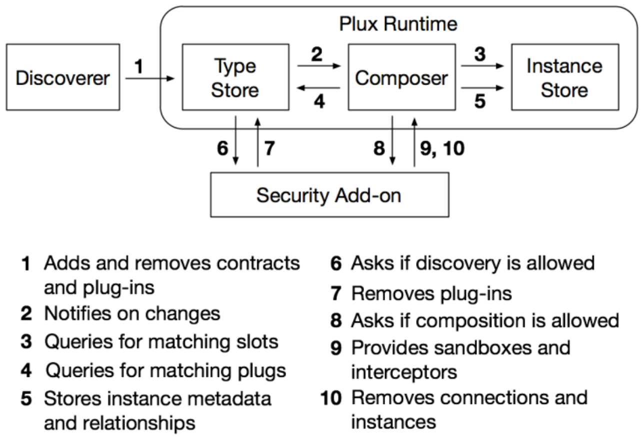

Plux Runtime
Discoverer
Type Store
Composer
Instance Store
Security Add-on
1
2
3
4
5
6
7
8
9, 10
1 Adds and removes contracts and plug-ins
2 Notifies on changes
3 Queries for matching slots
4 Queries for matching plugs
5 Stores instance metadata and relationships
6 Asks if discovery is allowed
7 Removes plug-ins
8 Asks if composition is allowed
9 Provides sandboxes and interceptors
10 Removes connections and instances

Figure 15: Architecture of the composition infrastructure

The *type store* maintains the metadata of slot definitions and extensions which are available for composition and notifies the composer about changes. When new metadata become available or when metadata are removed, the composer updates the program. In addition to that, the type store can be queried for contributors, e.g., by the composer when it tries to fill slots.

The composer assembles a program by matching requirements and provisions. It listens to changes in the type store and updates the program accordingly. If extensions become available or unavailable, it integrates or removes them and updates the composition state held in the instance store.

The *instance store* maintains the composition state of a program, i.e. the meta-objects of extensions, slots and plugs as well as the relationships between them. The instance store is also used by other tools which, for example, visualize the composition state and its changes during run time. Plux includes a visualizer tool which uses a notation similar to the one in the figures of this paper.

The *security add-on* is an optional part of the composition infrastructure. If the security add-on is installed, it blocks illegal composition operations, isolates plug-ins in sandboxes, and wires interceptors between extensions. Otherwise, if the security add-on is not installed, the composition infrastructure works without security restrictions.

3.2 Composition Process

The composition process is directed by the composer. On changes in the type store, the composer updates the program by matching slots and plugs. If a contributor becomes available in the type store, the composer queries the instance store for matching slots. A plug matches a slot if their names match. The composer plugs a matching plug into a slot if the slot definition is available, the plug implements the interface of the slot definition, and the plug provides values for the parameters declared by the slot definition. To plug a contributor means to instantiate it, add it to the instance store, and add a *plugged* relationship between the host and the contributor to the instance store. As the composer now treats the contributor itself as a host, it opens its slots and fills them with other contributors. In that way the composer continues the composition until all qualifying extensions are assembled.

Vice versa, if a contributor is removed from the type store, the composer queries the instance store for relationships containing the contributor's plugs. If it finds such relationships, it unplugs the contributor's instance. To unplug a contributor instance means to close its slots, to remove the *plugged* relationship from the instance store, to remove the contributor instance from the instance store, and to release it. Closing the contributor's slots causes the decomposition to be propagated, i.e. all contributors are unplugged from those slots as well.

Fig. 16 shows the steps performed by the composer when it activates a host and fills its slots. In Fig. 16.1, the extension *E1* is plugged into some host (not shown) but is not activated. In Fig. 16.2, the host of *E1* accesses the *Object* property of *E1*. Thus, the composer activates *E1*, i.e. it instantiates the associated extension object. As part of the activation, the composer opens the slot *S1* of *E1* (Fig. 16.3). Opening *S1* causes the type store to be queried for plugs matching *S1*. In our example, the composer finds *E2* with the plug *S1*. It creates an instance of *E2* and plugs it into *S1* of *E1* (Fig. 16.4). In Fig 16.5 the composition is completed and *E2* is in same state as *E1* in Fig 16.1.

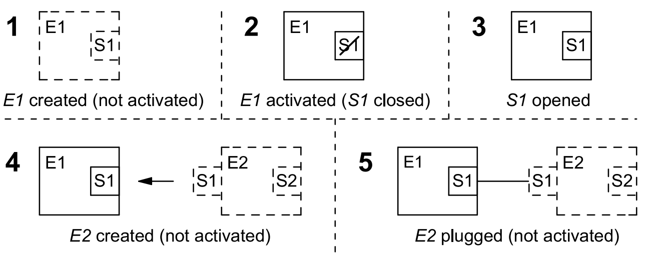

1
E1
S1
E1 created (not activated)
2
E1
S1
E1 activated (*S1* closed)
3
E1
S1
S1 opened
4
E1
S1
E2
S1
S2
E2 created (not activated)
5
E1
S1
E2
S1
S2
E2 plugged (not activated)

Figure 16: Steps of the Plux composition process

3.3 Runtime

The Plux runtime implements the composition infrastructure. To start a Plux program, one has to launch the runtime and provide a discoverer as well as the plug-in repository with the components that make up the program. At startup, the runtime activates its built-in *Startup* extension, which is the root for the Plux program. It has two slots: one for discoverers and one for applications.

To start the logger application from Section 2, we put the *Logger* contract, the *HostApp* plug-in, and the *ErrorLogger* plug-in into the plug-in repository, which is a file system folder. When we launch the runtime, we pass the folder and a file system discoverer as command-line arguments. The composer plugs the discoverer into the *Discoverer* slot of the *Startup* extension (Fig. 17). The *Startup* extension activates the discoverer, which discovers the contracts and the plug-ins from the provided folder. After the *HostApp* extension has become available in the type store, the composer plugs it into the *Application* slot of the *Startup* extension. The *Startup* extension activates *HostApp*, whereon the composer fills the *HostApp*'s *Logger* slot, i.e., it plugs the *ErrorLogger* extension.

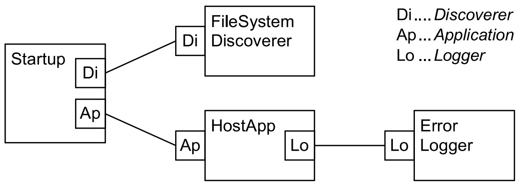


Figure 17: Plux runtime with startup extension and composed logger application

3.4 Security Add-on

The security add-on is integrated into the runtime. Fig. 15 in Section 3.1 shows how it interacts with the type store and with the composer. When a discoverer reports metadata, the type store asks the add-on if the discovery is allowed. If the security add-on blocks the discovery, the type store discards the reported metadata. When the composer instantiates an extension or connects two extensions, it firstly asks the add-on if the operation is allowed. If the add-on blocks the operation, the composer cancels the operation. If the add-on allows an instantiation, it can restrict the permission set for the created extension. The composer implements the restrictions by creating the extension in a sandbox with the provided permission set. If the add-on allows a connection, it can provide an interceptor which the composer wires between the two extensions.

Furthermore, the security add-on can become active to enforce restrictions by removing plug-ins, severing connections, or destroying extensions. This is necessary to respond to environmental changes, e.g., if a time slot passes, or the computer is connected to a network.

3.4.1 Security Devices

The security add-on comes with a library of predefined devices. It includes a composition interceptor, a call interceptor, and a sandbox. For scenarios which are not covered by predefined devices, developers can program custom devices.

The security add-on holds a chain of devices. For every operation request from Plux, the add-on consults all devices consecutively. If a device blocks the operation, the consultation ends and Plux cancels the operation. Only if all devices allow the operation, Plux proceeds with it (cf. Figure 18).

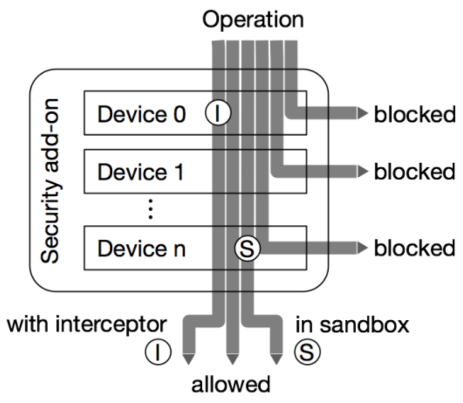


Figure 18: Chained security devices in the security add-on

If a device allows the instantiation of an extension, it can provide a permission set to limit what the extension can do. If multiple devices provide a permission set, the intersection of all permission sets will be used. If a device allows a connection, it can provide an interceptor. If multiple devices do so, the interceptors are chained.

3.4.2 Configuration

The security add-on reads the device configurations from configuration files. Multiple configuration files allow different stakeholders to specify restrictions, e.g., one file for the administrator and one for the user. A configuration consists of multiple restrictions. A restriction specifies a device name and multiple parameters which comprise a name and a value. The security add-on comes with an extensible device library. If a restriction cannot be realized with a predefined device, developers can program their own device.

3.5 Web Support

The web runtime of Plux provides the infrastructure for running plug-in-based web applications like the ones that were shown in Section 2.10. It extends the Plux desktop runtime with the following features:

- Multi-user support. For every user the web runtime maintains an individual set of plug-ins and an individual composition state.
- Distribution support. The web runtime can compose a web application from extensions running on different computers by plugging them remotely. It provides a distributed composition infrastructure that maintains the composition state of a distributed web application.

3.5.1 Multi-user Support

This section describes the infrastructure necessary to support multiple users with different sets of server-side extensions on a single web server. The handling of client-side extensions and distribution is shown in Section 3.5.2.

The web runtime supports multiple users by maintaining one *runtime node* per user on the web server. A runtime node comprises the infrastructure necessary to compose the web application for the corresponding user, i.e. a type store, an instance store, and a composer (Fig. 19). The type store maintains the type metadata for the user's extensions. The instance store maintains the user's composition state. The instances in a user's composition state are isolated from the composition states of other users, i.e., instances are not shared among users.

The composer is bound to the type store as well as to the instance store of a user. It assembles the extensions from the type store and stores the composition state in the instance store. For reasons of responsiveness, every runtime node is executed in a separate thread so that the composition for multiple users occurs concurrently.

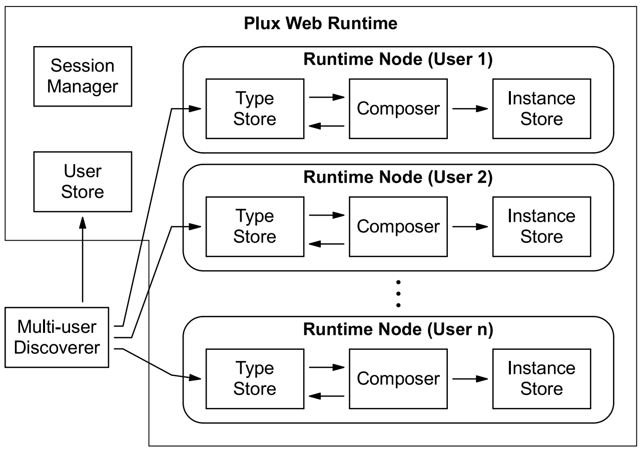


Figure 19: Architecture of the multi-user composition infrastructure

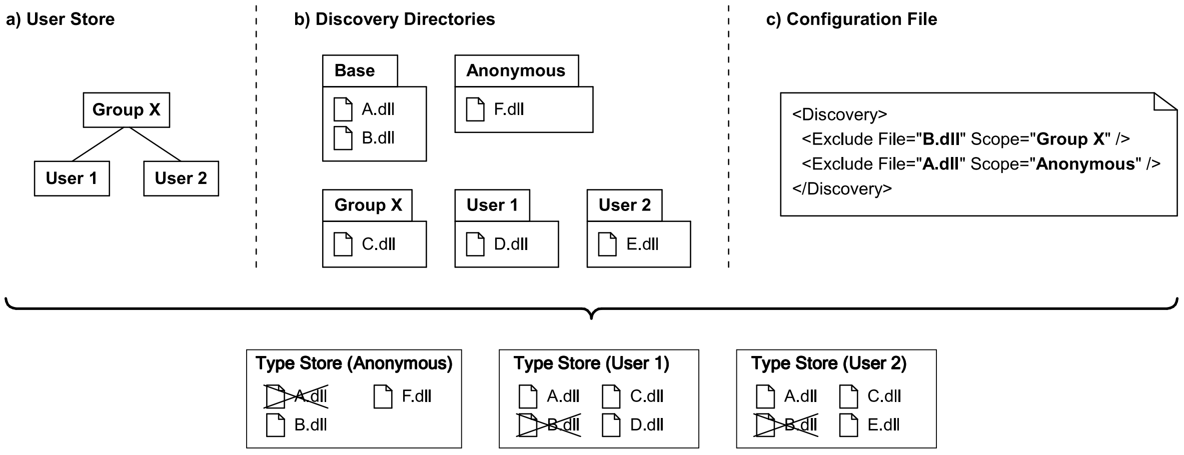
The server runs a *session manager* which creates a new runtime node when a user session begins, and releases the node when the session ends. During a session it reuses the runtime node for every request of this user.

Users can be hierarchically organized into groups. A user can be a member of multiple groups and a group can contain multiple users and groups. These membership relations are maintained in a *user store* (Fig. 20a).

Server-side extensions are kept in different directories on the server. For every user and for every group there is a directory with the user or group name, which contains the extensions that are specific to this user or group. In addition to that, there is a *Base* directory containing the extensions for all users as well as an *Anonymous* directory containing extensions for unauthenticated users (Fig. 20b).

The user's identity and his group memberships determine which plug-ins are available for him. For example, user 1 gets the plug-ins *A.dll* and *B.dll* from the *Base* directory, *C.dll* from the directory of group X, and *D.dll* from his own user directory. In other words, he inherits the plug-ins from *Base* and *Group X* and adds his own plug-ins.

Sometimes it is also necessary to exclude a plug-in from the inherited set of plug-ins. This is specified with a global *configuration file* like the one in Fig. 20c, which excludes *B.dll* for members of group X as well as *A.dll* for unauthenticated users.



```
<Discovery>
  <Exclude File="B.dll" Scope="Group X" />
  <Exclude File="A.dll" Scope="Anonymous" />
</Discovery>
```

Figure 20: The multi-user discoverer uses the user store, the discovery directories, and an optional configuration file to populate the users' type stores

3.5.2 Distribution Support

Distribution support allows the web runtime to execute the extensions on different computers and still to compose a coherent application from them. The web runtime supports distribution by remotely plugging extensions and by synchronizing the type stores and instance stores across multiple runtime nodes. For every user there is a runtime node in the web runtime and a runtime node on the client. We implemented client runtimes in different flavours, namely as plug-ins for Mozilla Firefox and Microsoft Internet Explorer, as a Silverlight application for rich client applications, and as a standalone application.

Remotely plugged extensions communicate via *extension proxies* on both sides of the line. An extension proxy consists of a proxy object, which communicates with the remote proxy, and a copy of the communication partner's meta-object, which represents the remote extension in the local instance store. The web runtime dynamically creates a contributor proxy on the host's runtime node, and a host proxy on the contributor's runtime node. These two proxies handle the remote communication, such that the host and the contributor need not care about distribution.

Fig. 21 shows a verbose and a compact notation for remotely plugged extensions. When the host *H* wants to call a method of the contributor *C*, it calls the corresponding method of the host-side *Proxy C*, which sends the call to contributor-side *Proxy H*. On the contributor-side, *Proxy H* calls the method of the contributor *C*. The results are sent back in the same way.

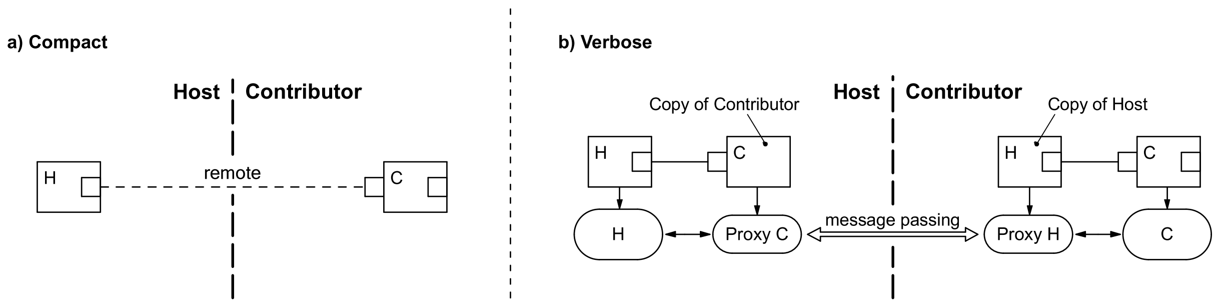


Figure 21: Compact and verbose notation for remotely plugged extensions

Every client as well as every Silverlight environment has its own runtime node with a replicated copy of the user's server-side composition state, i.e., the user's type store and instance store. In order to synchronize the composition states in the runtime nodes of the same user, the web runtime combines them into a common *user runtime*. Fig. 22 shows the user runtime of user 1 with two runtime nodes that are connected via *node coordinators*. The node coordinators synchronize the type stores and the instance stores, coordinate the composers, and provide a communication API for the proxies.

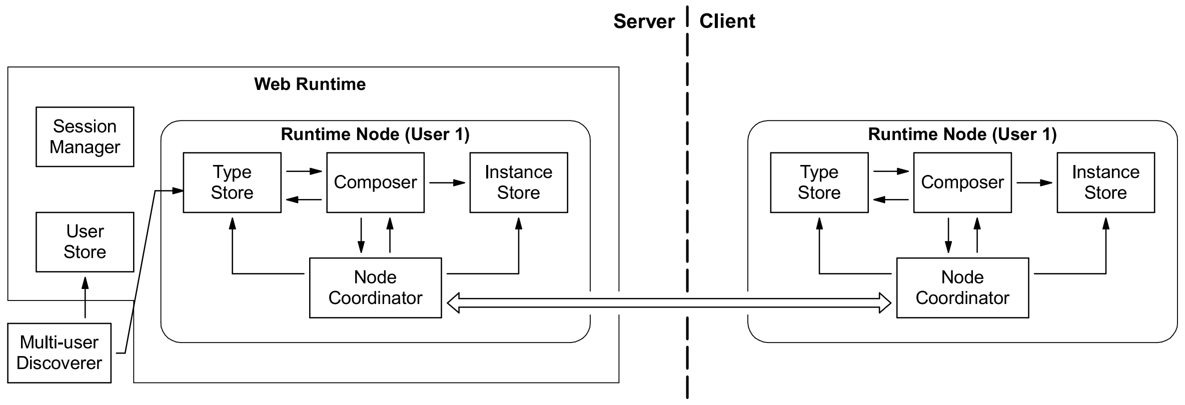


Figure 22: Distributed user runtime with interconnected runtime nodes

For consistency reasons, only one node per user runtime can be active at a time. In order to ensure this, the connected runtime nodes pass a token between each other. Only the node with the token can be active. Composition operations on this node do not lead to an immediate update of the other nodes. Only when the token is passed to some other node of the same user runtime, this node gets updated, i.e., it synchronizes its type store and its instance store with the data from the node that released the token. The node coordinator passes the token along when control is transferred to an extension that is executed on some other runtime node. Furthermore, a node coordinator can request the token with a broadcast to the connected runtime nodes. This happens, for example, when a user interaction initiates a composition operation on a runtime node that does not have the token.

4 Related Work

Plug-in frameworks have become quite popular recently. However, most of them were designed for building rich client applications and not for building web applications. Even frameworks that do allow building web applications lack support for per-user extensibility and customization.

Web programming platforms, on the other hand, support building web applications for multiple users, but without plug-in extensibility. Of course, there are web applications that support customization and per-user extensibility. However, such features are usually not provided by the platform, but have to be programmed by hand.

4.1 Plug-in Frameworks

The *Eclipse* Rich Client Platform [6] is a Java-based development platform for extensible desktop applications. Eclipse is based on the OSGi framework Equinox [5], which allows dynamic loading of components. Like Plux, Eclipse consists of a core that can be extended with plug-in components. The differences between Eclipse und Plux are: (1) The discovery mechanism for retrieving the metadata of components is an integral part of Eclipse, whereas in Plux it is an extension that can be replaced. (2) In Eclipse, the Java implementation of components is separated from their metadata, which are kept in XML files. In Plux, the metadata of components are specified with .NET attributes, which are placed directly in the source code. (3) In Eclipse, the hosts integrate their contributors themselves, whereas in Plux, programs are assembled automatically by the composer. (4) Eclipse maintains a registry of installed components, but only the hosts know which components they actually use. In contrast to that, Plux maintains a global composition state that keeps track of which components use which other components. (5) To support dynamic reconfiguration, an Eclipse host must provide both an implementation for integrating contributors at startup time and another one for integrating them dynamically at run time. In Plux, a single mechanism is used for integrating contributors both at startup time and at run time.

The Eclipse Rich AJAX platform RAP [11] allows building web applications using Eclipse plug-ins. A server-side Equinox environment [12] loads the Eclipse plug-ins on the web server and RAP delivers the user interface to the web browser using web technologies such as HTML and JavaScript. However, RAP does not maintain a global composition state, neither for a single user and even less for multiple users. It also does not support per-user customization.

SOFA 2 [13, 14, 15] is a system for building distributed component-based applications. The hierarchical component model distinguishes primitive and composite components. Primitive components are programmed, whereas composite components are declaratively composed from other primitive or composite components. The runtime environment allows users to transparently distribute the components across multiple nodes. The distributed components communicate with each other using automatically generated connectors. For communication, the connectors can use method invocation, message passing, streaming, or distributed shared memory. The runtime environment allows reconfiguring an application by adding, removing, and updating components at run time. The differences between Sofa and Plux are: (1) Sofa uses an architecture description language (ADL) to specify how a program should be composed from components. In contrast to that, Plux automatically establishes the desired composition by matching the requirements and provisions of components specified in their metadata. (2) Like Plux, Sofa can dynamically reconfigure an application. However, Sofa requires configurations to be statically specified in ADL, whereas Plux composes programs in a plug-and-play manner. (3) Sofa lacks support for extensible multi-user web applications, as it does not maintain the composition state for multiple concurrent users.

4.2 Web Programming Platforms

The Java Enterprise Edition (Java EE) [16] allows developing multi-tiered web applications using server and client components. Components on the server can be web components such as Java Servlets and JavaServer Faces as well as business components such as Enterprise JavaBeans. Components on the client can be application clients (rich-client applications) and applets. The differences between Java EE and Plux are: (1) Java EE applications are programmatically composed, whereas Plux applications are composed automatically in a plug-and-play manner. (2) Java EE lacks built-in support for per-user customization as well as for transparent remote plugging and unplugging. It also does not maintain a global composition state.

Web services [17] are a means to provide a public API for components over a network. Thus they can be used to build distributed component-based programs. One could compare a web service with a plug of a Plux component. In order to make a web service discoverable, it is registered in a public registry. A web service registry can be compared to a simplified Plux type store, containing only plugs. Consumers of web services typically integrate them using programmatic composition. Outside these components, the information about who uses which web service is unknown, i.e., the composition state is unavailable.

5 Summary and Future Work

In this paper we presented the dynamic plug-in framework Plux that targets both desktop and web applications. With Plux, every user can add his own components to a web application and has an individual composition state. The distributed composition infrastructure of Plux automatically composes a seamless application from components that can reside on different computers. Server-side components are installed and executed on the server. Client-side components are installed and executed on the client and can therefore use local resources there. Sandbox components are installed on the server, downloaded to the client on demand, and executed in a sandbox on the client. Components of any kind can be provided for a single user or for a group of users. The infrastructure transparently connects components that are executed either on the same computer or on different computers. The communication between remote components is handled by the infrastructure using dynamically generated proxies. The security concepts of Plux allow specifying composition restrictions based on the identity of the contributors, and to retrofit security using security devices.

Plux has been implemented under Microsoft .NET. However, its concepts are easily transferrable to any other platform (e.g. Java) that supports dynamic loading of components, interfaces, metadata annotations, as well as reflection. We have used Plux on various case studies, one of which is the time recorder web application that was used as an example throughout this paper.

Currently we are working on a layout manager for extensible user interfaces. Furthermore, in order to make web applications better scalable it must be possible to persist the composition of a user on the server and to restore it at the next round trip. Therefore, persistence is another aspect on our agenda.

Further information on Plux as well as a tutorial and a downloadable version of the framework can be found at http://ase.jku.at/plux/.

Literatur

[1] Wolfinger, R.: Dynamic application composition with Plux.NET: Composition model, composition infrastructure. Dissertation, Johannes Kepler University Linz, 2010.

[2] Mittermair, C.: Zerlegung eines monolithischen Softwaresystems in ein Plug-In-basiertes Komponentensystem. Master thesis, Johannes Kepler University Linz, 2010.

[3] Wolfinger, R., Reiter, S., Dhungana, D., Grünbacher, P., Prähofer, H.: Supporting runtime system adaptation through product line engineering and plug-in techniques. 7th IEEE Int Conf on Compos-based Softw Syst.:21.30, 2008. doi:10.1109/ICCBSS.2008.30

[4] Birsan, D.: On plug-ins and extensible architectures. ACM Queue 3(2):40–46, 2005. doi:10.1145/1053331.1053345

[5] OSGi Service Platform, Release 4. The Open Services Gateway Initiative. 2006, http://www.osgi.org. Accessed 28 July 2010

[6] Eclipse platform technical overview. Object Technology International, Inc, 2003. http://www.eclipse.org. Accessed 28 July 2010

[7] Boudreau, T., Tulach, J., Wielenga, G.: Rich client programming, plugging into the NetBeans platform. Prentice Hall International, 2007.

[8] ECMA international standard ECMA-335, Common Language Infrastructure (CLI), 4th edn, 2006.

[9] Jahn, M., Löberbauer, M., Wolfinger, R., Mössenböck, H.: Rule-based composition behaviors in dynamic plug-in systems. The 17th Asia-Pac Softw Eng Conf (APSEC 2010):80-89, 2010. doi: 10.1109/APSEC.2010.19

[10] Wolfinger, R., Löberbauer, M., Jahn, M., Mössenböck, H.: Adding genericity to a plug-in framework. 9th Int Conf on Gener Program and Compon Eng (GPCE 2010):93-102, 2010. doi: 10.1145/1868294.1868308

[11] The Rich Ajax Platform. 2010. http://www.eclipse.org/rap. Accessed 28 July 2010

[12] Server-side Equinox. 2010. http://www.eclipse.org/equinox/server. Accessed 28 July 2010

[13] Bures, T., Hnetynka, P., Plasil, F.: SOFA 2.0: Balancing advanced features in a hierarchical component model. 4th Int Conf on Softw Eng Res Manag and Appl (SERA 2006):40-48, 2006. doi:10.1109/SERA.2006.62

[14] Bures, T., Hnetynka, P., Plasil, F., Klesnil, J., Kmoch, O., Kohan, T., Kotrc, P.: Runtime support for advanced component concepts. 5th Int Conf on Softw Eng Res Manag and Appl (SERA 2007):337-345, 2007. doi:10.1109/SERA.2007.115

[15] Hnetynka, P., Plasil, F.: Dynamic reconfiguration and access to services in hierarchical component models. The 9th Int Symp on Compon-based Softw Eng (CBSE 2006):352-359, 2006. doi: 10.1007/11783565_27

[16] JSR 313: Java Platform Enterprise Edition 6 specification. 2007. http://jcp.org/en/jsr. Accessed 28 July 2010

[17] W3C Web services. 2002. http://www.w3.org/TR/ws-arch. Accessed 28 July 2010