Rule-based Composition Behaviors in Dynamic Plug-in Systems[1]

Markus Jahn, Markus Löberbauer, Reinhard Wolfinger, Hanspeter Mössenböck
Christian Doppler Laboratory for Automated Software Engineering
Johannes Kepler University Linz
4040 Linz, Austria
{jahn, loeberbauer, wolfinger, moessenboeck}@ase.jku.at

Plug-in frameworks facilitate the development of customizable and extensible software, yet they often lack support for flexible and dynamic (re)configuration. We have created Plux.NET, a novel plug-in framework for plug-and-play composition. In Plux, a composer replaces programmatic composition with automatic composition. Components just specify their requirements and provisions using metadata, and the composer assembles the components guided by that metadata. This paper introduces *rule-based composition behaviors*, which are a means for controlling the composition process declaratively. Behavior rules constrain the composer by preventing certain operations or by triggering new ones. They help to establish a rule-conformant composition state. Thereby, Plux supports developers in declarative and rule-based composition in order to minimize programming effort.

Component-based software; Plug-in architecture; Run-time adaptation; Software reuse; Rule-based system

I. 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 can thus be tailored to the specific needs of individual users. Despite the success of plug-in frameworks so far, current implementations still suffer from several deficiencies:

(a) Weak automation. Host components have to integrate extensions programmatically instead of relying on automatic composition. Furthermore, plug-in frameworks usually have no control over whether, how or when a host looks for extensions.

(b) Poor dynamic reconfigurability. Host components integrate extensions only at startup time whereas dynamic addition and removal of components is either not supported or requires special actions in the host.

(c) Limited Web support. Plug-in frameworks primarily target rich clients or application servers. Although some frameworks extend the plug-in idea to web clients, they are still limited in customizability and extensibility: They neither support individual plug-in configurations per user, nor do they allow the integration of plug-ins executed on the client.

Over the past few years we developed a plug-in framework called Plux.NET which tries to solve the problems described above [1][2][3][4][5]. This paper deals with a fresh view of issue (a) and describes how the automatic composition mechanism of Plux can be improved by assigning rule-based composition behaviors to components.

A major advantage of Plux over established plug-in systems (like OSGi [6] or Eclipse [7]) is how the composition process is performed: Plug-in systems usually have two kinds of components. Those that provide functionality (*contributor* components) and those that want to use the provided functionality (*host* components). Contributors register their functionality at a global repository while host components listen to changes in the repository and use the functionality they need. The problem in this solution is that every host component has to implement a mechanism that looks for its contributors itself. This results in code duplication and may lead to an inconsistent composition implementation. A further problem with this technique is that there is usually no run-time information about the actual composition state (e.g., which host is using which contributor component). Only the host component itself possibly knows which components it uses, but there is no information about the global composition state. Therefore, it is almost impossible to automatically perform dynamic reconfiguration of an application for a specific user task. If the system does not know the actual composition state, it simply cannot reconfigure it.

To solve the problems mentioned above the composition logic of Plux is not implemented in every host component but in a central *composer* of the Plux runtime. Contributors specify which functionality they provide and hosts specify which functionality they need. The composer uses this information to connect hosts to matching contributors when they are added to a repository and disconnects them when they are removed. All connections are stored by the runtime and can be queried for dynamic reconfiguration. Components can also react to events which are raised when components are connected or disconnected. By these means, Plux minimizes coding effort and maximizes support for dynamic reconfiguration.

Although automatic composition is sufficient for many situations, developers sometimes need more explicit control over which components should be connected to a host and

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

which should not. Therefore, Plux allows disabling the automatic composition for individual components or even for the entire application. In this scenario, component implementations can use the Plux runtime API to look for available extensions and compose or decompose them programmatically. This is even helpful in combination with automatic composition. Components can listen to composition events and can react, for example, by rejecting certain contributors or by replacing existing components with new ones.

The disadvantage of controlling the composition by code is that it leads to programming overhead in the components, like in other plug-in systems. In our case studies we detected, that there were many equivalent pieces of code in different components. This means code duplication, higher error probability and components flooded with code that has nothing to do with their actual tasks. Although Plux stores the composition state also in the case of manual composition, our goal is to control the composition process without implementing code in various components.

This paper describes how to achieve controlled automatic composition without programming overhead in the components. In our approach the composition process is controlled by applying rule-based composition behaviors to the components.

Our research was done in cooperation with BMD Systemhaus GmbH, a company offering line-of-business software in the ERP domain. ERP applications consist of many different features that can either be used together or in isolation, thus being an ideal test bed for our automatic plug-in composition approach.

The paper is organized as follows: Section II describes the plug-in framework Plux from the view of component developers. Section III explains how the automatic composition process of Plux is performed. Section IV shows some frequently used scenarios where composition behaviors come into play. Section V covers the implementation of composition behaviors. Section VI compares our work to related research. The paper closes with a summary and a prospect of future work.

II. THE PLUX.NET FRAMEWORK

Plux.NET [1] is a .NET-based plug-in framework that allows composing applications from plug-in components. It consists of a thin core that has slots into which extensions can be plugged. Plugging does not require any programming. The user just drops a plug-in (i.e., a DLL file) into a specific directory, where it will be automatically discovered and plugged into one or several matching slots. Removing a plug-in from the directory will automatically unplug it from the application. Thus, adding and removing plug-ins is completely dynamic allowing applications to be reconfigured for different usage scenarios without restarting the application.

Plug-in components (so-called *extensions*) can have *slots* and *plugs*. A slot is basically an interface describing some expected functionality. A plug belongs to a class implementing this interface and thus providing the required functionality. Slots, plugs and extensions are specified declaratively using .NET attributes. Thus the information that is necessary for composition is stored directly in the metadata of interfaces and classes and not in separate XML files as for example in Eclipse [7].

Let's look at an example. Assume that some host component wants to print log messages with time stamps. The logging should be implemented as a separate component that plugs into the host. We first have to define the slot into which the logger can be plugged.

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

The *slot definition* is a C# interface tagged with a `[SlotDefinition]` attribute specifying the name of the slot (`Logger`). For simplicity reasons we use simple names (e.g. `"Logger"`) in our example. However, in complex applications names can be prefixed with for example a company's domain name.

Slots can have parameters defined by `[ParamDefinition]` attributes. In our case we have one parameter `Verbosity` of type `int`, which is to be filled by the contributor and used by the host. The verbosity specifies the level of messages which are printed by the logger. Next, we are going to write an extension that fits into the Logger slot:

```
[Extension("ConsoleLogger")]
[Plug("Logger")]
[Param ("Verbosity", 1)]
public class ConsoleLogger: ILogger {
  public void Print(string msg) {
    Console.WriteLine(msg);
  }
}
```

An extension is a class tagged with an `[Extension]` attribute. It has to implement the interface of the corresponding slot (here `ILogger`). The `[Plug]` attribute defines a plug for this extension that fits into the Logger slot. The `[Param]` attribute assigns the value `1` to the parameter `Verbosity` which means that the logger is interested in messages with verbosity level greater or equal to 1.

Finally, we implement the host, which is another extension that plugs into the `Application` slot of the Plux `Startup` extension. The slot's interface `IApplication` specifies the method `Start` which is called by the `Startup` extension when an `Application` is plugged into the `Application` slot. `MyApp` creates a new Thread in `Start` which logs a message with the actual time stamp every second. In doing so, `MyApp` iterates over each plugged logger extension and examines the loggers' verbosity levels. If the level is large enough the message is printed.

The `Logger` slot of the extension `MyApp` is defined with a `[Slot]` attribute. This attribute also specifies an event handler method `LoggerPlugged` that will be called when an extension is plugged into this slot. `LoggerPlugged` prints a notification with the name of the logger which was plugged. In more sophisticated applications the `Plugged` event is typically used for giving the host a chance to retrieve information about the plugged extension.

```
[Extension("MyApp")]
[Plug("Application")]
[Slot("Logger", OnPlugged="LoggerPlugged")]
public class MyApp: IApplication {

  public void Start() {
    new Thread(Run).Start();
  }
  void LoggerPlugged(CompositionEventArgs args) {
    Print("Logger plugged: "
            + args.Plug.Extension.Name, 1);
  }
  void Run() {
    while (true) {
      Print(DateTime.Now.ToString, 0);
      Thread.Sleep(1000);
    }
  }
  void Print(string msg, int verbosity) {
    foreach(Plug p in
              Slots["Logger"].PluggedPlugs) {
      int v = (int) p.Params["Verbosity"].Value;
      if (v >= verbosity) {
        ILogger l = (ILogger) p.Extension.Object;
        l.Print(msg);
      }
    }
  }
}
```

This is all we have to do. If we compile the interface `ILogger` as well as the classes `ConsoleLogger` and `MyApp` into DLL files and drop them into the plug-in directory everything will fall into place. The `FileSystem Discoverer` which comes with the Plux framework will discover the extensions `MyApp` and `ConsoleLogger`, and the composer of Plux will plug `MyApp` into the `Application` slot of the Plux `Startup` extension and `ConsoleLogger` into the `Logger` slot of `MyApp` (see Figure 1).

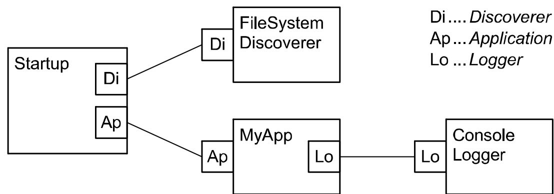

Figure 1. Composition architecture of the logger example

Plux offers a light-weight way of building plug-in systems. Plug-ins are just classes tagged with metadata. They are self-contained, i.e., they include all the metadata necessary for discovering them and plugging them together automatically. There is no need for separate XML configuration files. The example also shows that Plux is event-based. Plugging, unplugging and other actions of the runtime core raise events to which the programmer can react. The implementation of Plux follows the plug-in approach itself. For example, the discovery mechanism that monitors the plug-in directory is itself an extension and can therefore be replaced with some other way of discovery.

III. THE AUTOMATIC COMPOSITION PROCESS

The logger example in Section II showed how to develop extensions for Plux applications that get composed automatically by the composer of the plug-in framework. This section describes the composition infrastructure of Plux and shows how the composer works. The composition infrastructure comprises the components `Discoverer`, `Type Store`, `Composer` and `Instance Store` (see Figure 2).

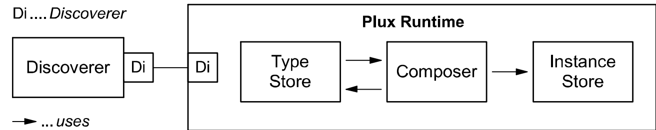

Figure 2. The composition infrastructure of the Plux Runtime

The `Discoverer` is responsible for discovering new plug-ins and for detecting when plug-ins are removed. It creates the type information for new extensions and adds them to the `Type Store`. The `Discoverer` is implemented as an extension itself (see Figure 1). Thus, the discovery mechanism can be replaced or extended by further extensions (e.g. by a discoverer for network plug-ins or by a discoverer that reads the extensions' metadata from XML files or from a data base).

The `Type Store` stores the type information for available extensions. It notifies the `Composer` when new extensions are added or when they are removed. In addition to that, it is the source for extension types in which the `Composer` finds plugs that match opened slots.

The `Composer` is the component that actually performs the automatic composition. It creates new instances of extensions, activates them, opens their slots and finally plugs matching plugs and slots together. Furthermore, it is responsible for automatic decomposition. If an extension is not needed any more and gets unplugged, the composer checks if the extension is still in use by other extensions. If not, it destroys both the extension and all its contributors that are not used any more. Even though, automatic composition is sufficient in most situations, extension developers also can control the composition process programmatically.

The `Instance Store` stores the actual composition state, which was composed by the `Composer`. It contains the run-time information for extensions, including their state and the state of their slots. The `Instance Store` is not only used by the `Composer` but also by other tools which, for example, show the actual composition state during run time. This provides a very useful support for extension developers. The representation of the composition state displayed by our tools is similar to the representation in the figures of this paper.

We will now look at how the composition process is performed. After the `Discoverer` has discovered a new extension, it is stored in the `Type Store`. From now on the type information of the extension (its name, plugs, slots, etc.) is available and the `Composer` gets notified. If the `Composer` finds slots that match the extension's plugs, it creates an instance of the extension and plugs it into those slots. At this time the extension is *created*, but not yet *activated*. This means that only an envelope with the extension's metadata is created but not the actual extension object. As soon as some-

one tries to access the extension object, it gets instantiated. Then the extension is *activated*. Thus, host extensions can already retrieve the metadata of a plugged contributor before its actual extension object is instantiated. This lazy instantiation speeds up the startup of Plux applications.

Figure 3 shows our representation of *not activated* and *activated* extensions. The *verbose* representation explicitly shows the extension's envelope with its extension object. However, in this paper we use the *compact* representation where the extension object is not shown explicitly. Instead, we use dashed versus solid lines to distinguish between *not activated* and *activated* extensions.

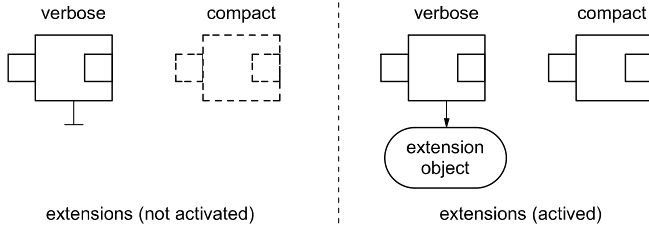

Figure 3. Verbose and compact representations for not activated and activated extensions

As soon as an extension has been activated, the `Composer` opens the extension's slots and looks for matching plugs. Then it creates the extensions for the matching plugs and plugs them into the just opened slots. When all newly activated extensions are composed, the composer is idle until the next modification of the `Type Store` or the next user action that triggers a composition action occur.

Figure 4 shows the steps in which the composition process is performed. Please note that in steps 4 and 5 the extension *E2* is in the same state as extension *E1* in step 1.

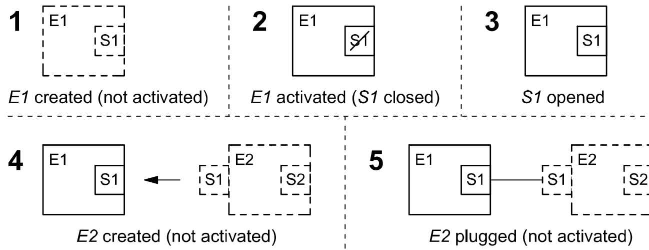

Figure 4. Steps of the Plux composition process

Due to the lazy activation of extensions and the resulting lazy composition, Plux composes only the minimal set of extensions that is in use. Furthermore, if extensions are not in use anymore they can be unplugged, where the `Composer` works in a similar way as a garbage collector. If an extension is not plugged anymore its contributors get unplugged and it gets destroyed. Since this continues until all unused extensions are destroyed, Plux keeps the actual composition state as small as possible. This reduces the complexity of applications as well as their resource consumption. Benchmarks have shown that our approach scales well. Even in large applications with hundreds of plug-ins the discovery and composition takes less than a second.

A further concept of the Plux composition model is *selecting* connections between plugs and slots. We use this when a host has multiple contributors and uses only some of them at a certain point in time. In this case, we denote the contributors that are currently in focus by using *selected* connections. In that way, the composition model provides more detailed information about the composition state, which is useful for a better understanding of an application or for adjusting the state of a user interface. The concept of selecting connections is used in the example below.

IV. Controlling the Automatic Composition with Rule-based Behaviors

Using the automatic composition process of Plux, extensions just need to be tagged with the right metadata for being plugged together without any programming effort. However, in some situations developers do not want every plug to get connected to every matching slot automatically. For this reason, Plux provides an API to disable automatic composition and to perform composition manually by code in the extensions. Unfortunately, this bloats the components with code that has nothing to do with their actual tasks. Furthermore, this code cannot be reused by other extensions. Consequently, this approach ends up in code duplication which is error prone and may lead to inconsistencies. After all, the automatic composition process of Plux supports developers in implementing extensions with minimal coding effort. Therefore, we want to achieve both: using the automatic composition process and controlling it in order to satisfy specific needs of extension developers.

Our approach for achieving customized automatic composition is to specify a certain composition behavior either for individual slots or for the composer as a whole. There are predefined behaviors, which concern various steps of the composition process (i.e., opening slots, creating and activating extensions, plugging plugs and selecting connections), but developers can also implement custom behaviors. The logic of a behavior is specified by the extension developer using a rule. Currently we have two kinds of rules. Self-contained rules and rules that bind the composition state to the user interface, so that a user can control the actual composition state by user interface actions.

The example below (Figures 5-9) deals with some patterns that we found in our case studies. Due to our behavior mechanism, we could eliminate a lot of duplicated code by applying behaviors with an appropriate rule instead. In the figures below, we only show extensions that are relevant for explaining our behavior approach, while extensions such as `Startup` or `Discoverer` are hidden.

The sample application consists of the extension `Workbench` which has a slot `View` for view extensions (`Email`, `Payroll`) and a slot `Container` for a container extension (`MDIContainer`), which is responsible for displaying the plugged views. Since views can only be displayed in a container, they are useless if no container extension is plugged. As it should not be possible to plug useless extensions, the `View` slot has to be kept closed, until a container is plugged. Figure 5 shows the desired scenario. In step 1 no container is plugged, so the `View` slot has to be kept closed. As soon as a

container extension is plugged into the `Container` slot, the `View` slot has to be opened so that the composer can plug matching extensions into this slot (step 2). Finally, in step 3 the composer plugs the views into the `Workbench` and consequently the `MDIContainer` extension displays them as child windows in the application window.

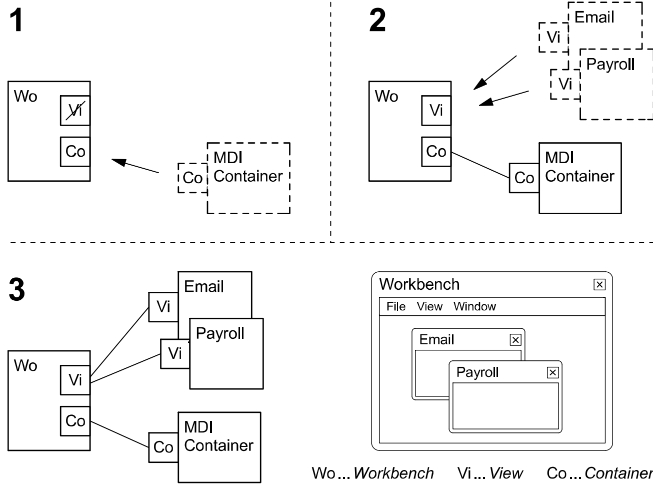

Figure 5. Applying a PlugBehavior with an OpenSlotRule that opens a specified slot (View), if at least one exension is plugged into the slot to which the behavior is applied (Container)

To achieve the scenario of Figure 5 we apply a `PlugBehavior` to the slot `Container`. This behavior notifies its rule when an extension was plugged into the slot to which the behavior was applied. As a rule for this behavior we use the `OpenSlotRule` that opens a specified slot (`View`) if at least one contributor is plugged into the behavior's slot (`Container`).

```
[Extension]
[Plug("Application")]
[Slot("View", OnPlugged="... )]
[Slot("Container", OnPlugged="... )]
class Workbench : ExtensionBase, IApplication {

  Workbench() {
    Slots["Container"].Behaviors.Add(
      new PlugBehavior(
        new OpenSlotRule( Slots["View"] ) ) );
  }
  ...
}
```

As the `PlugBehavior` also notifies its `OpenSlotRule` when an extension gets unplugged, the rule reacts on this event as well. If no extension is plugged in the `Container` slot anymore, the rule closes the `View` slot. Since behaviors even get notified before a composition action will happen, their rules can already react before the action is actually performed. So the `OpenSlotRule` can close the `View` slot before the last container is unplugged and the constraint, that the `View` slot is only open if at least one extension is plugged into the `Container` slot, always holds (see Figure 6).

Next we show how behaviors can be reused with different rules, and that it can be useful to apply multiple beha-

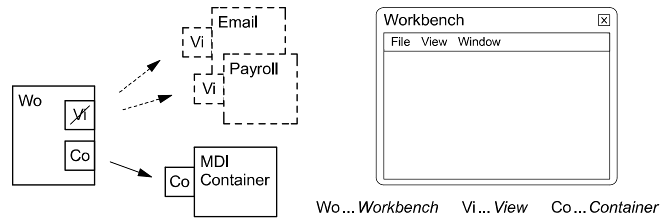

Figure 6. Applying a PlugBehavior with an OpenSlotRule: Before the last extension is unplugged from the behavior slot (Container) a specified slot (View) gets closed and its extensions get unplugged

viors to a single slot. In our example the `Workbench` has a slot for the container extension, which is responsible for arranging the views, and which can be exchanged at run time. However, the `Workbench` can only deal with a single container at the same time. Therefore, we want to make sure that the number of plugged containers is limited to one. A solution for that is to apply a further `PlugBehavior` to the `Container` slot, but this time using a `ReplaceRule`. This rule unplugs previous extensions from this slot, as soon as new extensions get plugged and the maximum number of plugged extensions is already reached. The maximum can be set through a parameter of the rule's constructor.

```
...
  Workbench() {
    Slots["Container"].Behaviors.Add(
      new PlugBehavior(
        new OpenSlotRule( Slots["View"] ) ) );

    Slots["Container"].Behaviors.Add(
      new PlugBehavior( new ReplaceRule(1) ) );
  }
...
```

If multiple behaviors are applied to a single slot, developers have to keep in mind that the execution order of behaviors can matter. The execution order is the same as the order

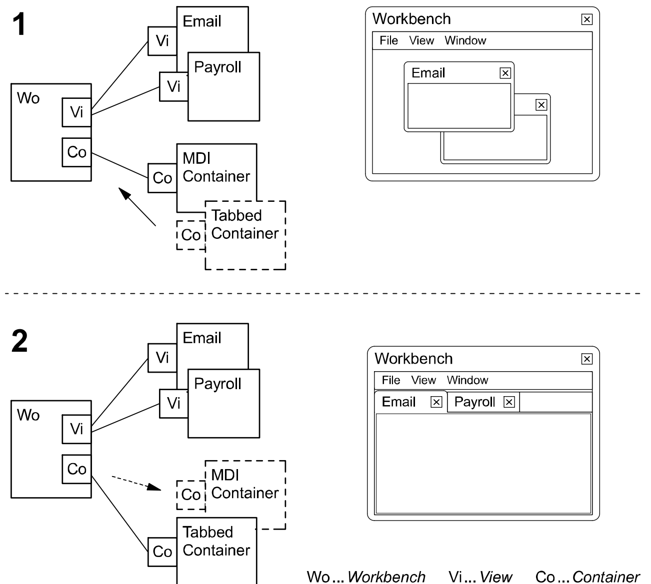

Figure 7. Applying a PlugBehavior with a ReplaceRule to ensure that there is only one extension plugged into the Container slot at the same time

in which the behaviors are applied. In our example the behaviors are independent, thus the order does not matter.

Figure 7 shows how a `TabbedContainer` extension gets plugged into the `Container` slot. As an `MDIContainer` was already plugged there and the maximum number of extensions in this slot was set to one in the `ReplaceRule`, the rule springs into action and unplugs the `MDIContainer` extension. Since the `MDIContainer` is not plugged any more, it automatically gets destroyed by the composer. Finally, the `TabbedContainer` extension is plugged into the `Container` slot and displays the views in a tabbed panel.

In the same way as behaviors can be reused with different rules, also rules can be reused with different behaviors. Let's look at an example for reusing the `ReplaceRule`, but this time in combination with a `SelectBehavior`. This behavior deals with the selection of plugged plugs. As explained in Section III, the concept of selecting connections between slot and plugs allows us to set the focus on one or several extensions from the set of all extensions plugged into a slot. In our example we use this to select the view extension that has the current focus. Since only one view can have the focus at the same time, every time when a new view gets selected the `SelectBehavior` with the `ReplaceRule` removes the previous selection.

```
...
  Workbench() {
    ...
    Slots["View"].Behaviors.Add(
      new SelectBehavior( new ReplaceRule(1) ) );
  }
...
```

In Figure 8.1 the `Payroll` view is in focus, i.e., the connection from the `Workbench` to the `Payroll` extension is selected. This is represented as a filled knot in the middle of the connection. In Figure 8.2 the `Email` view gets the focus. Therefore, the connection from `Workbench` to `Email` gets selected and the `ReplaceRule` removes the previous selection. A problem in this solution is that the extension developer has to implement the functionality to keep the composition model in sync with the currently focused view. Actually this means that we have to write code for controlling the composition state. Since we want to avoid the disadvantages of composition code in the components, we apply a `SelectBehavior` to the `View` slot with a `FocusRule` (instead of the `ReplaceRule`) that binds the currently focused view to the selected connection. This rule reacts when the focus changes to a different view and adapts the composition state accordingly. Vice versa, when the composition state changes, the `FocusRule` makes sure that the selection is adapted to the new state. In order to allow the `FocusRule` to control the focus of the views, it has a reference to the current container, whose interface specifies a `FocusChanged` event and a method to focus a view. Thus, the `FocusRule` is independent from the container extension that is actually plugged into the `Container` slot.

Due to the fact that behaviors can bind the application state to the composition model, various tools can control the application through a common model. Furthermore, if a tool, for example, persists the actual composition state before an application is closed, it can restore it at the next startup, and the application will be in the same state as before without any additional effort.

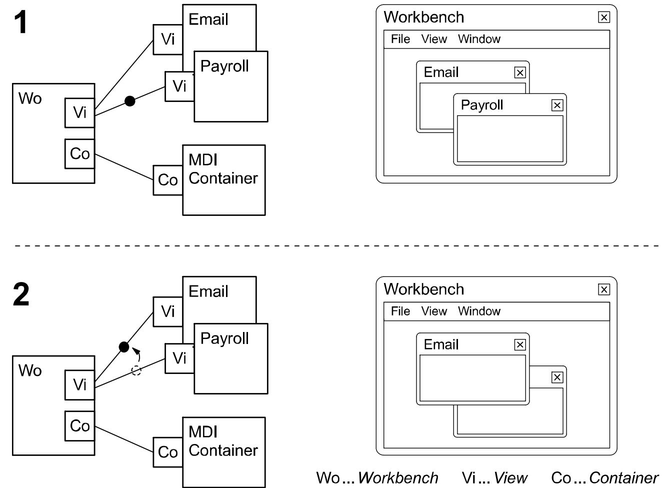

Figure 8. Keeping the selection in sync with the focused view by applying a SelectBehavior with a FocusRule

The next scenario shows another elegant example for using behaviors in order to link the composition state to the user interface. We use them for automatically generating menu items that control the currently opened views as well as the currently focused view. Figure 9 shows how behaviors do that just by modifying the composition state and by listening to its changes. Steps 1 to 6 show the current composition state and the corresponding states of the menus *View* and *Window*. The *View* menu contains menu items for all available views that can be opened. If the user clicks on one of these menu items, the corresponding view is opened and the menu item is displayed with a check mark. A further click on a checked menu item closes the view and unchecks the menu item. The *Window* menu contains menu items for all opened views, where the menu item for the focused view is checked. Clicking on an item in the *Window* menu focuses the clicked view.

To set up the above scenario we apply the `ActivateBehavior` and the `SelectActivatedBehavior`, both with the `MenuRule`, to the `View` slot. The `ActivateBehavior` handles the activation of plugged extensions and is used for the *View* menu. The `SelectActivatedBehavior`, which is similar to the `SelectBehavior`, deals with already activated extensions and is used for the *Window* menu. The `MenuRule` generates menu items for all extensions that are involved in a composition action initiated (or reacted on) by the corresponding behavior. For the `ActivateBehavior` these are all extensions that get plugged. For the `SelectActivatedBehavior` these are all activated extensions that get selected. The menu to which the menu items are added is specified as a parameter of the `MenuRule`. As soon as a user clicks on a menu item, the according composition action is performed and the menu item gets checked. Let’s look at the example in Figure 9:

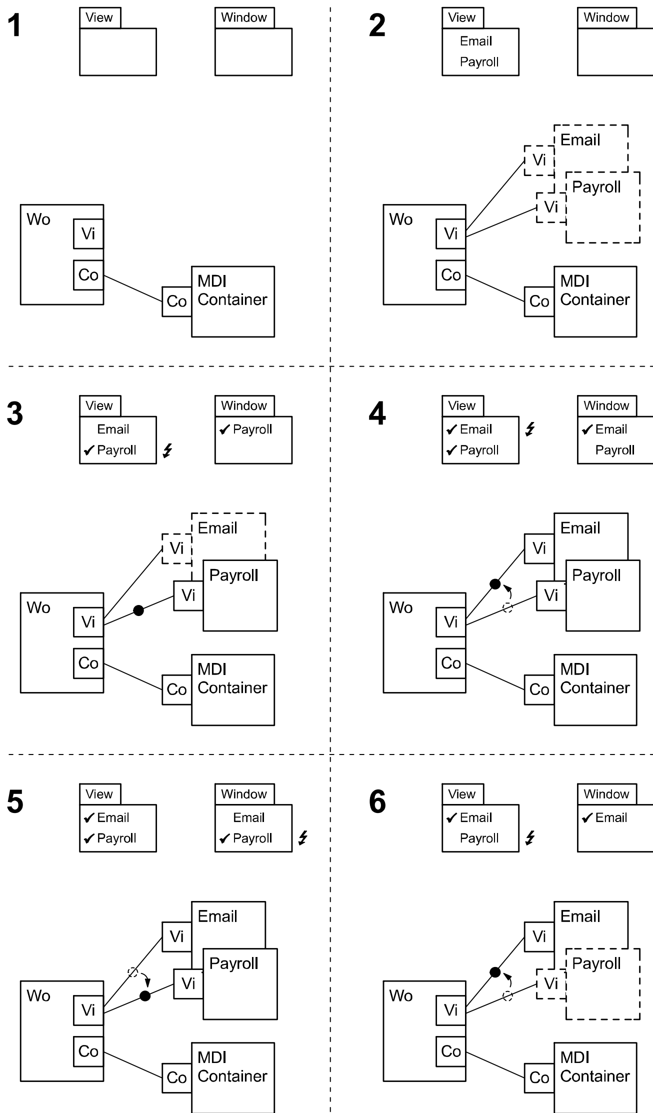

Figure 9. Generating menu items out of the composition state.

Step 1. No views are plugged into the `View` slot. Therefore, the *View* menu is empty. Consequently no activated views can be selected, so the *Window* menu is empty as well.

Step 2. As soon as views are plugged they are shown in the *View* menu. Since the plugged views (`Email`, `Payroll`) are *not activated* yet, the *Window* menu is still empty.

Step 3. By clicking on the *Payroll* menu item in the *View* menu the `Payroll` view gets activated and is therefore opened as MDI child window. When a view is opened it automatically gets the focus, so the connection to the `Payroll` view gets selected. Since the `Payroll` view is activated and selected, the *Payroll* item is checked in both menus.

Step 4. By clicking on the *Email* menu item in the *View* menu the `Email` view gets activated and selected so that both views are visible now and the `Email` view is selected. So the *Window* menu has items for both views now, but the checked one is the *Email* menu item.

Step 5. Next, the user clicks on the *Payroll* menu item in the *Window* menu, thus changing the selection to the `Payroll` view. The *Window* menu now shows the *Payroll* menu item as being checked.

Step 6. Finally, the user clicks the *Payroll* menu item in the *View* menu. This deactivates the `Payroll` extension which causes the `Payroll` view to be closed and the `Email` view to be selected. The *Payroll* menu item is unchecked in the *View* menu and removed from the *Window* menu.

The state of the menu items is triggered by the current composition state, but the menu items can also be used to control which views are opened and which view is focused. This works because of the binding between the user interface and the composition state, which is accomplished by behaviors and does not require any programming overhead. Therefore, the menus and the views are always in sync, even if the composition state is changed by a tool or by another behavior. Newly added views do not need to register themselves in order to be integrated; they just need to specify a `View` plug. Everything else is taken care of by the composition framework using the specified behaviors. Thus, Plux applications are easily extensible.

In our case study, the `Workbench` extension also has an `Action` slot into which other menu items can be plugged. Menu items such as *New*, *Open*, or *Exit* are implemented as extensions with an `Action` plug. They can be added to the menu strip just by dropping the corresponding extensions into the plug-in repository. We also have additional behavior rules that bind the composition state to further user controls such as buttons, combo boxes and list boxes. Thus, software developers can build highly dynamic user interfaces just by applying behaviors to certain slots.

This section described how pre-defined behaviors and rules can be used to control the automatic composition process of Plux. However, developers also can implement their own behaviors and rules which are tailored to their special needs. The following section explains how behaviors and rules are implemented.

V. Implementation of Behaviors and Rules

Behaviors can react to composition actions. As they are either applied to the composer itself or to a certain slot, they can either react to all composition actions of the entire application or only to actions that modify the composition state of the slot to which they are applied (called the *behavior slot*). Since most behaviors deal with certain steps of the composition process, they can specify a set of *composition handlers* that will be called during the composition. For a composition action (e.g., opening a slot or plugging a plug) behaviors can register up to three handlers that are called by the composer:

The first handler is called to ask applied behaviors, if the composer is allowed to perform the upcoming action. Thus, a behavior can cancel a composition action. This is, for example, used in behaviors with the `SecurityRule` that only allows plugging extensions of a certain manufacturer.

The second handler is called directly before the composition action is performed. This gives behaviors a chance to update the composition state in order to prepare it for the

upcoming action. For example, behaviors with the OpenSlotRule can close their target slot before the last extension gets unplugged from the behavior slot.

The third handler is executed after the composition action was performed. This is the most frequently used handler. For example, a behavior with the OpenSlotRule opens its target slot as soon as an extension is plugged into its behavior slot.

Since behaviors are defined for different composition or decomposition actions, they register different handlers. For example, the SelectBehavior registers the composition handlers CanSelect, Selecting, and Selected as well as the decomposition handlers CanDeselect, Deselecting, and Deselected.

A behavior not only reacts to a certain composition action but also has some logic attached to it. The logic of a behavior is specified as a *rule*. Rules are implemented as strategies for behaviors, because most rules are useful in many different composition actions. For example, in Section IV we used the ReplaceRule and the MenuRule with the PlugBehavior and the SelectBehavior.

As it is the rules that handle the logic of behaviors, behaviors simply forward handler calls to their rules. Rules do not know which composition action they are actually handling, so they have three generic handlers for the three handler types described above. Every rule implements the generic handlers CanCompose, Composing and Composed that are independent of the actual composition action, and they also implement the handlers CanDecompose, Decomposing and Decomposed for decomposition.

Most rules also need to modify the composition state, so they must be able to perform composition actions, but these actions depend on the behavior to which the rule is attached. Therefore, rules can delegate a Compose or a Decompose call back to their behavior, which translates it into the appropriate composition action. For example a SelectBehavior translates Compose into Select and Decompose into Deselect.

Figure 10 shows an example of how a SelectBehavior and a ReplaceRule work together when the composer notifies the behavior that a connection will be selected shortly. First, the composer calls the behavior's Selecting handler, which forwards the notification to the ReplaceRule by calling its Composing handler. Assuming that the maximum number of connections is already selected, the rule calls back the behavior's Decompose handler and passes the plug to be deselected as a parameter. The Decompose handler of the SelectBehavior finally calls Deselect on the behavior slot and thus deselects the formerly selected plug.

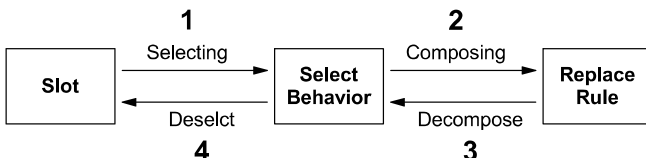

Figure 10. SelectBehavior forwards the Selecting call to the generic Composing call of its rule. The callback Decompose from the rule gets translated into a Deselect composition action, which is performed on the slot to which the behavior is applied.

Some rules require information about which candidates are available for composing or decomposing, and the behaviors have to provide this information. This is frequently used in rules that bind the composition state to the user interface (e.g., MenuRule or ComboBoxRule). The candidates for a composition action vary depending on the behavior. For example, the candidates for the PlugBehavior are all plugs that fit into the behavior slot, while the candidates for the SelectBehavior are all plugs that are currently plugged into the behavior slot.

Figure 11 shows the interfaces and classes that are needed for the rule-based behavior mechanism. Slots as well as the composer itself can have an arbitrary number of behaviors, where each behavior must implement the interface IBehavior. This interface specifies all handlers that can be called by the composer. A *flag* property specifies for every behavior which handlers will be registered at the composer and will therefore be called during the corresponding composition action.

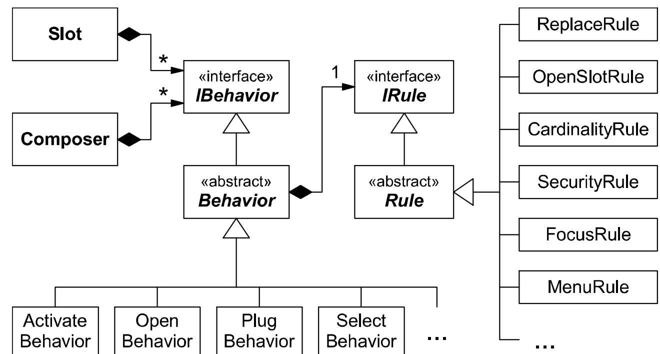

Figure 11. Class diagram for rule-based behavior implementations

The class Behavior is an abstract implementation of IBehavior that has a reference to its rule as well as empty handler implementations, so that the actual behaviors only need to override the handlers they need. The specific behavior classes inherit from Behavior, implement the handlers they need, and use the *flags* property to specify which handlers these are.

Every rule has to implement the interface IRule that specifies the generic composition and decomposition handlers as well as methods that can be used by the behavior to add and remove the candidates for its composition action. Additionaly, the rule has two properties that allow behaviors to register the handlers that translate the Compose and the Decompose calls back to the appropriate composition action. The abstract class Rule provides empty implementations for the handlers in IRule so that the rule implementations just need to override the handlers they need. The following listings show, how the ReplaceRule and the SelectBehavior are implemented.

ReplaceRule has a queue that contains all items which are already composed. The type parameter T denoting the item type depends on the behavior to which the rule is attached. For example, in combination with the SelectBehavior the composed items are the selected plugs, whereas with the ActivateBehavior the composed items are the activated extensions. The field capacity is set by the constructor and specifies the maximum number of items that are

allowed to be composed simultaneously. Furthermore, the constructor sets the property `CandidatesRequired` that specifies if the behavior has to provide the candidates for its composition action. As `ReplaceRule` does not require the candidates for the behavior's composition action, it is set to `false`.

`ReplaceRule` overrides only the handler methods `Composing`, `Composed` and `Decomposed`, because other handlers are not needed in this rule. The `Composing` handler checks if the maximum number of composed items is already reached. If so, it calls `Decompose` (inherited from `Rule`), which forwards the call back to the attached behavior. The behavior translates the call into the appropriate decomposition action (e.g., `Deselect` or `Deactivate`). The parameter of this call is the first item in the composition queue (`queue.Peek()`). The `Composed` handler enqueues newly composed items. Since `Composing` is always called before `Composed`, it is guaranteed that the maximum number of items in the queue is never exceeded. In the `Decomposed` handler the rule removes the item that is decomposed from the queue:

```
class ReplaceRule<T> : Rule<T> {
  readonly Queue<T> queue = new Queue<T>();
  readonly int capacity;

  ReplaceRule(int capacity) {
    this.capacity = capacity;
    CandidatesRequired = false;
  }
  override void Composing(T item) {
    if (queue.Count == capacity)
      Decompose(queue.Peek());
  }
  override void Composed(T item) {
    queue.Enqueue(item);
  }
  override void Decomposed(T item) {
    queue.Remove(item);
  }
}
```

`SelectBehavior` inherits from the abstract class `Behavior`. The rule for the behavior is set in the base class constructor. The constructor of `SelectBehavior` sets the `BehaviorFlags` property that specifies which handlers should be registered at the composer. The method `Bind` is called when the behavior gets applied to a slot. It is used to check if the rule requires candidates for the behavior's composition action. If so, the method `AddCandidate` is called for each plugged plug to add it to the rule as a candidate for selecting. Additionally, a `Plugged`-event handler and an `Unplugged`-event handler are registered at the behavior slot, so that they can update the candidates when plugs get plugged or unplugged.

`SelectBehavior` overrides all handlers that are called by the composer during the composition action *select* (e.g., `CanSelect` and `Selecting`) and forwards their calls to the appropriate generic handlers (`CanCompose` and `Composing`). Furthermore, it implements the handlers `Compose` and `Decompose` that are called by the rule and forwards these calls to `Select` and `Deselect`, which are implemented in the base class:

```
class SelectBehavior : Behavior<Plug> {

  SelectBehavior(IRule<Plug> rule) : base(rule) {
    BehaviorFlags = BehaviorFlags.CanSelect
                  | BehaviorFlags.Selecting
                  | BehaviorFlags.Selected
                  | BehaviorFlags.CanDeselect
                  | BehaviorFlags.Deselecting
                  | BehaviorFlags.Deselected;
  }
  override void Bind(Slot slot) {
    base.Bind(slot);
    if (Rule.CandidatesRequired) {
      foreach(Plug plug in slot.PluggedPlugs);
        Rule.AddCandidate(plug);
      slot.Plugged += Slot_Plugged;
      slot.Unplugged += Slot_Unplugged;
    }
  }
  override bool CanSelect(SelectEventArgs e) {
    return Rule.CanCompose(e.Plug);
  }
  override void Selecting(SelectEventArgs e) {
    Rule.Composing(e.Plug);
  }
  override void Selected(SelectEventArgs e) {
    Rule.Composed(e.Plug);
  }
  ...
  override void Compose(Plug plug) {
    Select(plug);
  }
  override void Decompose(Plug plug) {
    Deselect(plug);
  }
  void Slot_Plugged(PlugEventArgs e) {
    Rule.AddCandidate(e.Plug);
  }
  void Slot_Unplugged(PlugEventArgs e) {
    Rule.RemoveCandidate(e.Plug);
  }
}
```

VI. RELATED WORK

Rule-based systems are a means to store, manipulate, and interpret knowledge. Rule-based programming (also known as logic programming, pioneered by Prolog [8]) expresses the logic of a computation without describing its control flow. In other words, it describes what a program should do, rather than how to do it. A rule-based system consists of a rule set (i.e., the knowledge), an inference engine that processes the input and the rule set, and an interface to provide the input. The goal of rule-based programming is to use knowledge and deduction to solve problems by deriving logical consequences. In contrast to that, imperative programming specifies algorithms explicitly.

In plug-in frameworks, rule-based composition is new. Other plug-in frameworks such as OSGi [6], Eclipse [7] or

NetBeans [9] rely on programmatic composition, where the programmer connects components explicitly. This means that they follow the imperative paradigm. In Plux, the composer replaces programmatic composition with automatic composition. Components just declare their requirements and provisions using metadata. The composer assembles components guided by that metadata. Behaviors extend the declarative mechanism of Plux with rules to guide the composer. Such rules can be used to declaratively influence the composition, either of an individual slot or of a combination of several slots. Rules can prevent or trigger composition operations.

If we understand Plux as a rule-based system, we can identify the following parts: The composition state and the rules represent the knowledge. The composer is the inference engine which processes composition events as input. It performs the logical consequences, i.e. it triggers or prevents composition operations. The composer's interfaces are the type store and the instance store.

Behaviors and rules follow well-known design patterns [10]. A behavior is an *Observer* for composition actions on slots and plugs. Any such action triggers the registered behaviors. A rule is a *Strategy* that is attached to a behavior. A strategy is an exchangeable algorithm for solving a certain problem. In this case, it solves the problem of reacting to composition actions in some user-defined way.

VII. Conclusion

In this paper we presented an approach for rule-based composition and its integration into the plug-in framework Plux. Our approach solves the problem of controlling the automatic composition process of Plux without programmatic composition in the components.

The Plux composer automatically connects components that provide certain services to other components that require them. In doing so, the composer follows a default rule: it simply connects all components where the provisions match the requirements. By using rule-based behaviors, developers can add further rules that tailor the composition process to their specific needs.

Behaviors react to composition actions of the composer, where the logic of a behavior is implemented as an exchangeable rule. Since we provide many predefined behaviors and rules for common composition patterns, developers can control the composition process of their applications by just applying these behaviors in combination with the appropriate rules (in the sense of a black-box framework). However, developers also can implement their custom behaviors and rules.

The major benefit of using rule-based behaviors instead of programmatic composition is that behaviors can be reused in many situations. Thus, behaviors lead to less programming overhead and are therefore less error-prone. Furthermore, as behaviors are separated from the components' implementation, they do not bloat them, and more importantly: they can be exchanged—even at run time.

Since we use behaviors to bind the application state to the composition state of Plux, various independent tools can control the application by modifying the underlying composition model. Thus, the application state can be adapted to specific user tasks, or it can be persisted and restored without any additional effort.

In the scenarios of this paper we only showed some general purpose behaviors and rules that are reusable in many applications. However, as developers can implement their own rules and behaviors, they can develop more complex rules that are tailored to their applications. For example, developers may apply application-specific rules for composition behaviors that evaluate conditions such as: if A is connected to B and B is not connected to C then disconnect A from B.

In the future, we will develop further mechanisms for specifying rule-based behaviors. For example, our business partner BMD configures applications through databases. Therefore, they need to apply behaviors and rules that are specified in a database.

References

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

[2] M. Jahn, R. Wolfinger, and H. Mössenböck, “Extending web applications with client and server plug-ins,” Software Engineering 2010, SE 2010, Paderborn, Germany, February 22-26, 2010.

[3] R. Wolfinger, S. Reiter, D. Dhungana, P. Grünbacher, and H. Prähofer, ”Supporting runtime system adaptation through product line engineering and plug-in techniques,” 7th IEEE International Conference on Composition-Based Software Systems, ICCBSS 2008, Madrid, Spain, February 25-29, 2008.

[4] R. Wolfinger, M. Löberbauer, M. Jahn, and H. Mössenböck, “Adding genericity to a plug-in framework,” submitted to GPCE'10.

[5] R. Wolfinger, D. Dhungana, H. Prähofer and H. Mössenböck, “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.

[6] OSGi service platform, Release 4. The open services gateway initiative, http:/ /www.osgi.org, July 2006.

[7] Eclipse platform technical overview, Object technology international, Inc., http:/ /www.eclipse.org, February 2003.

[8] A. Colmerauer, P. Roussel, “The birth of Prolog,” The second ACM SIGPLAN conference on History of programming languages, p. 37-52, 1992.

[9] T. Boudreau, J. Tulach, G. Wielenga, “Rich Client Programming, Plugging into the NetBeans Platform,” 2007.

[10] E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns, Elements of reusable object-oriented software. Addison-Wesley 1995.