Testing the Composability of Plug-and-Play Components

A Method for Unit Testing of Dynamically Composed Applications

M. Löberbauer, R. Wolfinger, M. Jahn and H. Mössenböck
Christian Doppler Laboratory for Automated Software Engineering
Institute for System Software, Johannes Kepler University, Linz, Austria
{loeberbauer, wolfinger, jahn, moessenboeck}@ase.jku.at

***Abstract*—Software systems, which are dynamically composed from plug-and-play components, allow users to adapt an application to the working scenario at hand. While the testing of individual components is well understood, there are no systematic techniques that test if components can be assembled in arbitrary orders. This paper introduces a method and a tool for testing the dynamic composability of component-based software systems. It is based on Plux.NET, a plug-in platform for plug-and-play composition of .NET applications.**

I. Introduction

Plug-and-play components allow users to assemble customized applications without configuration or programming effort. This can be used to adapt feature-rich applications to the needs of individual users. *Dynamically* composed plug-and-play components allow users to reconfigure an application on the fly by swapping sets of components. Thus users can align the application with the working scenario at hand.

Plug-and-play composition requires additional testing in order to check the composability of the components. In this paper, composability means *dynamic* composability, i.e., components can be added and removed at run time. We call a component a *host* if it uses other components; and we call it a *contributor* if it provides a service to other components. A contributor is composable if it can be integrated and removed dynamically. A host is composable if its contributors can be added and removed dynamically and in any order.

As dynamic composition is a relatively new approach in component frameworks, it is not covered by current test methods and tools. We have created Plux.NET, a plug-in framework that supports dynamic plug-and-play composition. From our case studies we learned that many composability deficiencies only show up if contributors are added in a particular order, or when contributors are removed. In this paper, we present a method for unit testing which determines the test cases that are relevant for checking the composability as well as a tool to automate the testing.

Our research was conducted in cooperation with our industrial partner BMD Systemhaus GmbH. BMD is a medium-sized company offering a comprehensive suite of enterprise applications, e.g., customer relationship management (CRM), accounting, production planning and control. As most of their customers use only a fraction of the suite, customized products are an essential part of BMD's strategy.

This paper is organized as follows: Section II describes the Plux framework. It explains how components declare their requirements and provisions with metadata, how the components are discovered, and how an application is assembled from components. Section III gives a motivating example with a composability deficiency, discusses why existing test methods do not find the problem and outlines requirements for a solution. Section IV describes our method for composability testing, a tool for test automation, and the integration of the tool into Plux. Section V describes how existing component frameworks handle testing. Section VI finishes with a conclusion and an outlook to future work.

II. The Plux.NET Framework

The Plux.NET framework (Plux) was created to allow developers to build applications using dynamic plug-and-play composition [1]. It enables extensible and customizable applications that can be reconfigured without restarting the application. Together with our industrial partner we applied Plux to the CRM product of BMD [2]. By allowing dynamic addition and removal of CRM features, we support a set of new usage scenarios, e.g., on-the-fly product customization during sales conversations as well as incremental feature addition for step-by-step user trainings [3].

The unique characteristics of Plux are the *composer*, the *event-based programming model*, the *composition state*, and the exchangeable *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 the host queries a service registry and integrates its contributors itself. *Automatic composition* means that the components just declare their requirements and provisions using metadata; the composer uses these metadata to match requirements and provisions, and connects matching components automatically. The hosts can react to *events* sent by the composer during composition. Plux also maintains the current *composition state*, i.e., it stores which hosts use which contributors. As hosts can retrieve the composition state from Plux, they do not need to store references to their contributors. *Discovery* is the process of detecting 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, thus making the mechanism replaceable.

The following subsections cover those characteristics in more detail; Section III shows their implications on testing using a motivating example.

A. Metadata

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

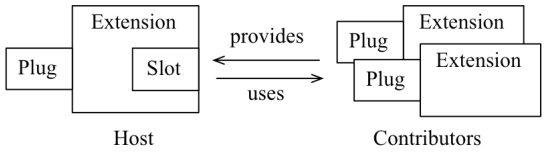


Fig. 1 Extensions with slots and plugs.

The slots and plugs are identified by names. A plug matches a slot if their names match. If so, the plug can be connected to the slot. The host uses connected contributors over a defined interface. This interface is specified in a *slot definition*. A slot definition has a unique name and optionally parameters; contributors must set the parameters and hosts can retrieve them. The names of slots and plugs refer to the expected and provided slot definitions respectively.

The means to provide metadata is customizable in Plux. The default mechanism extracts metadata from .NET custom attributes. Custom 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 *ParamDefinition* attribute to declare required parameters, the *Extension* attribute to tag classes that implement components, the *Slot* attribute to declare requirements in hosts, the *Plug* attribute to declare provisions in contributors, and the *Param* attribute to declare provided 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 should use a parameter to declare if it wants to print 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);
}
```

Fig. 2 Definition of the *Logger* slot.

Next, we are going to write logger contributors. Fig. 3 shows the logger for errors. The logger for warnings is implemented the same way (not shown).

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

Fig. 3 Error logger as a contributor for the Logger slot.

Finally, we implement the application that uses the loggers (Fig. 4). Since it is a host for loggers, it has a *Logger* slot. However, it is also a contributor to the Plux core, so it has an *Application* plug. At startup, the Plux core creates contributors for the *Application* slot. The complete implementation of the application is shown in Subsection D.

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

Fig. 4 Application host with the *Logger* slot.

B. Discovery

In order to match requirements and provisions, Plux needs the metadata of the extensions. The discovery is the part of Plux which extracts the metadata. By default, it extracts them from the custom attributes stored in the .NET assembly files. However, since the discovery mechanism is an extension itself, it is replaceable and can retrieve the metadata also from a database or from a configuration file. The collected metadata is stored in the *type store* of Plux.

Discoverers listen to changes in a component repository, i.e. they detect if extensions are added or removed. The order in which a discoverer detects changes depends on its implementation. For example, the order in which an attribute discoverer reads metadata from assembly files might differ from the order in which a text-file-based discoverer would read the same metadata from a text file. The motivating example in Section III shows how this order affects the composability of components.

C. Composer

The composer assembles applications by matching slots and plugs. For this purpose, it observes the type store. If a contributor becomes available in the type store, the composer integrates it into the application. Similarly, if a contributor is removed from the type store the composer removes it from the application.

Integrating a contributor means that the composer instantiates it and connects its plug with the matching slot of a host. This is repeated for each matching slot in the application. When the composer connects a plug with a slot, it notifies the host and stores the connection. The instances and their connections make up the *composition state*. The part of Plux that stores the composition state is called the *instance store*.

Removing a contributor means that the composer disconnects the plug from the slot and releases the contribu-

tor. When the composer disconnects a plug from a slot, it notifies the host and removes the connection from the instance store.

The described mechanism where the composer reacts to changes in the type store is called *automatic composition*. In addition to that, applications can be assembled with *programmatic composition* in which the composer is controlled by extensions. For example, host extensions can integrate contributors using API calls, a script interpreter can assemble an application from a script, or a serialization extension can restore a previously saved composition state. Thus the sequence in which the composer makes connections differs depending on whether automatic or programmatic composition is used.

D. Composition State

The composition state can be retrieved from the instance store. For every instantiated extension, the instance store holds the extension's meta-objects of its slots and plugs as well as a reference to the corresponding .NET object (Fig. 5). For every slot, the instance store holds the information about which plugs are connected.

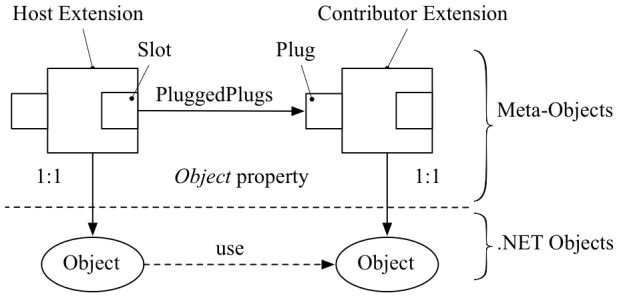


Fig. 5 Meta-objects for instantiated extensions in the instance store.

Fig. 6 describes the host of Fig. 4 in more detail showing how meta-objects can be used by an application. 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 that does the rest of the work.

```
[Extension]
[Plug("Application")]
[Slot("Logger")]
public class HostApp : IApplication {
  Slot loggerSlot;
  public HostApp(Extension e) {
    loggerSlot = e.Slots["Logger"];
    new Thread(Run).Start();
  }
  void Run() {
    while(true) {
      string msg; LoggerKind kind;
      Work(out msg, out kind);
      foreach(Plug p in loggerSlot.PluggedPlugs) {
        if(kind == (LoggerKind) p.Params["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 */
  }
}
```

Fig. 6 Application host using logger contributors.

In the *Run* method, the host does its work and then uses the connected loggers to print a message. It retrieves the loggers via the *PluggedPlugs* property of the logger slot. For each logger, it checks the logger kind using the parameter *Kind*. Finally, it retrieves the .NET objects of the selected loggers and prints the message. As Plux instantiates contributors only on demand, i.e. when the host accesses the extension's *Object* property, loggers of other kinds are not instantiated in this example; only their meta-objects exist.

Additionally, the host can react to events that the composer sends when it connects or disconnects contributors. This is appropriate for hosts that want to react on added or removed contributors immediately. Fig. 7 shows a modified version of our host from Fig. 6. It uses the *Slot* attribute to register event handlers 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="LoggerPlugged",
  OnUnplugged="LoggerUnplugged")]
public class HostApp : IApplication {
  ...
  void LoggerPlugged(CompositionEventArgs args) {
    Extension e = args.Plug.Extension;
    ILogger logger = (ILogger) e.Object;
    logger.Print("Logger plugged: " + e.Name);
  }
  void LoggerUnplugged(CompositionEventArgs args) {
    ...
    logger.Print("Logger unplugged: " + e.Name);
  }
  void Run() { ... }
  void Work(...) { ... }
}
```

Fig. 7 Modified application host reporting connected contributors.

Thus there are two ways for retrieving the connected contributors: (i) Retrieve the plugs connected to a slot from the instance store; (ii) React to the *Plugged* events sent by the composer. Both mechanisms are affected by the order in which components are discovered (cf. Subsection B) as well as by the order in which programmatic composition connects the components (cf. Subsection C).

Section III shows a motivating example that demonstrates how the various orders can cause failures in the parts of the host that retrieve connected contributors. Therefore these parts should be subject to composability testing.

III. Motivating Example

Assume that we want to create an application that copies data from a source to a sink. Sources and sinks should be implemented as contributors that plug into the application. Therefore, the copy application has two slots: the *Source* slot and the *Sink* slot (Fig. 8).

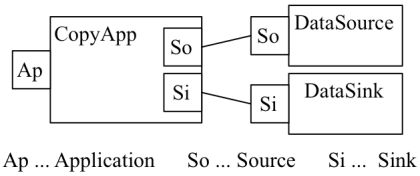


Fig. 8 Copy application with source and sink contributor.

The fact that the copy application (host) has two slots and that there is a dependency between them (i.e., the source contributor is supposed to be connected before the sink contributor) makes the host vulnerable for ordering errors.

Fig. 9 shows an error-prone implementation of the host. When the composer connects a sink to the host, a *Plugged* event is raised, and the event handler *CopyData* starts copying data from the source to the sink. *CopyData* incorrectly assumes that there is already a source connected, and will fail if a sink gets connected before a source, because the *PluggedPlugs* collection is empty in this case.

```
[Extension]
[Plug("Application")]
[Slot("Source")]
[Slot("Sink", OnPlugged = "CopyData")]
public class CopyApp : IApplication {
  void CopyData(CompositionEventArgs args) {
    Extension self = args.Slot.Extension;
    ISource source = (ISource) self.Slots["Source"]
        .PluggedPlugs[0].Extension.Object;
    ISink sink = (ISink) args.Plug.Extension
        .Object;
    sink.Write(source.Read());
  }
}
```

Fig. 9 Copy application (error-prone implementation).

Assume that the developer of the copy application works with a specific discoverer and with automatic composition. Coincidentally, the order in which the discoverer adds the extensions corresponds to the order assumed by the host. Thus the problem does not show up. However, the host will fail, if it is used in a setup where the discoverer adds the extensions in a different order.

For comparison, Fig. 10 shows a robust implementation of the same host using programmatic composition. To avoid that a sink gets connected before a source, the host opens the sink slot manually. In Plux, the composer connects contributors only when a slot is open. By default, the composer opens slots automatically. In this example, however the host disables this behavior by setting the *AutoOpen* property of the slot attribute to *false*. The sink slot is opened programmatically as soon as a source gets connected. Vice-versa, the host closes the sink slot when a source gets disconnected; this causes an already connected sink to be unplugged. In summary, this causes a toggling behavior where the host opens and closes a sink slot depending on whether a source is connected.

```
[Extension]
[Plug("Application")]
[Slot("Source", OnPlugged = "OpenSink",
    OnUnplugged = "CloseSink")]
[Slot("Sink", AutoOpen = false,
    OnPlugged = "CopyData")]
public class CopyApp : IApplication {
  void OpenSink(CompositionEventArgs args) {
    Extension self = args.Slot.Extension;
    self.Slots["Sink"].Open();
  }
  void CloseSink(CompositionEventArgs args) {
    Extension self = args.Slot.Extension;
    self.Slots["Sink"].Close();
  }
  void CopyData(...) { ... }
}
```

Fig. 10 Copy application (robust implementation).

To avoid the programming effort, one can achieve the same toggling behavior by applying *rule-based behaviors*. Rule-based behaviors are a declarative means to control the composer. For an extensive description see [9].

With rule-based behaviors we can also solve another not yet mentioned problem of our example. Namely, the composer might connect multiple source contributors to the source slot. In this case, the host will incorrectly ignore all sources but the first one. To correct this, we can apply a specific behavior to the source slot that limits the cardinality of the slot to a single contributor.

The above example shows the need for composability testing, because common integration testing does not consider dynamic composition and does not detect problems caused by different orders of composition.

IV. Plux Composability Test Method and Tool

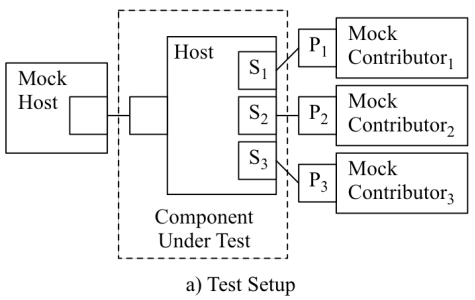
The Plux Composability Test Method (PCTM) is a method for testing the dynamic composability of plug-and-play components. The goal is to reveal problems caused by different orders of composition. PCTM is a dynamic black-box test method: Dynamic, because it runs the composer repeatedly and varies the order in which the slots of a host are filled. Black-box, because it looks only at the declared metadata of components and ignores their source code. We do this, because we want to detect index out of bounds errors and null pointer errors in the parts of the host that access the composition state. In this paper we focus on the composability of hosts, whereas contributors are left for future work.

A. Approach

PCTM tests each component (testee) individually using the following steps: (i) Generate a mock host with slots for every plug of the testee. (ii) Connect the testee with the generated host. (iii) Generate a mock contributor for every slot of the testee. (iv) Determine a set of composition sequences by permuting the order in which the slots of the testee should be filled. (v) Select a sequence from the set and connect the mock contributors with the testee's slots (if open) in the specified order; monitor the testee for errors. (vi) Repeat step v for each sequence in the set.

All mock components can be generated automatically. Since slots and plugs refer to a slot definition, which is basically an interface, the method signatures of the mock components can be generated in such a way that they conform to those interfaces. Note that we are not interested in generating mock components that do real work, but only in detecting null pointer and index out of bounds errors that result from unexpected composition orders.

Fig. 11a. shows the testee, how it is connected to the mock host, and how its slots are filled with mock contributors. Fig. 11b. shows the set of sequences to test.



S1	S2	S3
P1	P2	P3
P1	P3	P2
P2	P1	P3
P2	P3	P1
P3	P1	P2
P3	P2	P1

b) Sequences

Fig. 11 Test setup and possible composition sequences.

Let us revisit error-prone implementation of the example from Section III (Fig. 9) and apply the following steps in order to test the composability of the copy application (Fig. 12): (i) Generate a mock host with an *Application* slot. (ii) Connect the *CopyApp* to the mock host. (iii) Generate a mock data source contributor and a mock data sink contributor. (iv) Determine the set of composition sequences: { Source, Sink } and { Sink, Source }. (v) Connect contributors in the order { Source, Sink }. (vi) Repeat step v with the order { Sink, Source }. In step vi the test procedure reveals the out of bounds error. If we for comparison apply the same test procedure to the robust implementation of the host (Fig. 10), it does not find any errors (as expected).

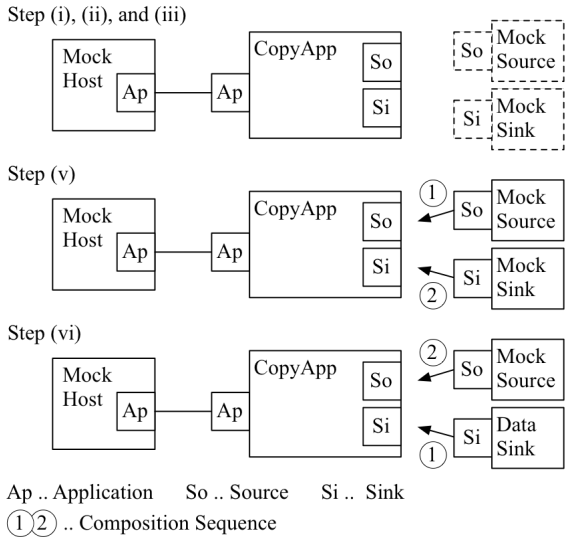


Fig. 12 PCTM test procedure for copy application example.

B. Tool

The Plux Composability Test Tool (PCTT) implements the Plux Composability Test Method as a Plux extension. To start the tests, PCTT connects to the Plux core as an application. To add the generated mock extensions to the type store, it also connects to the core as a discoverer. PCTT has a *Mock* slot to integrate the mock hosts which it generates (Fig. 13a).

When PCTT runs the tests it disables automatic composition and uses programmatic composition to connect the testee with the generated mock components. If applied to the copy application from Section III, PCTT performs the following steps: (i) Generate a component *MockHost* with a *Mock* plug and a slot for integrating *CopyApp* via its *Application* plug; add *MockHost* to the type store. (ii) Generate the contributors *MockSource* and *MockSink* (Fig. 13b). (iii) Connect *MockHost* to *PCTT*. (iv) Connect *CopyApp* to *MockHost*. (v) Determine the valid composition sequences: { *MockSource*, *MockSink* } and { *MockSink*, *MockSource* }. (vi) Apply the first test sequence { *MockSource*, *MockSink* } and monitor *CopyApp* for exceptions (Fig. 13c). (vii) Disconnect *MockSource*, *MockSink*, and *CopyApp*. As the state of *CopyApp* might change during composition, we use a new instance for each test run. (viii) Repeat steps iv to vii with the next test sequence until all sequences have been tested.

Like unit test frameworks, PCTT runs all tests, no matter whether exceptions occur or not. Using the terminology of JUnit [10], a single test sequence corresponds to a test method, all test sequences for a testee correspond to a test case, and all test cases for a component repository correspond to a test suite.

When PCTT runs a test suite it logs all exceptions. Using the graphical user interface of PCTT the user can review the results. Test cases and test methods that revealed errors are highlighted. For each test method the user can retrieve the detected exceptions.

For the test suite of the copy application the tool highlights the second test method with the sequence { *MockSink*, *MockSource* } and reveals the detected index out bounds exception.

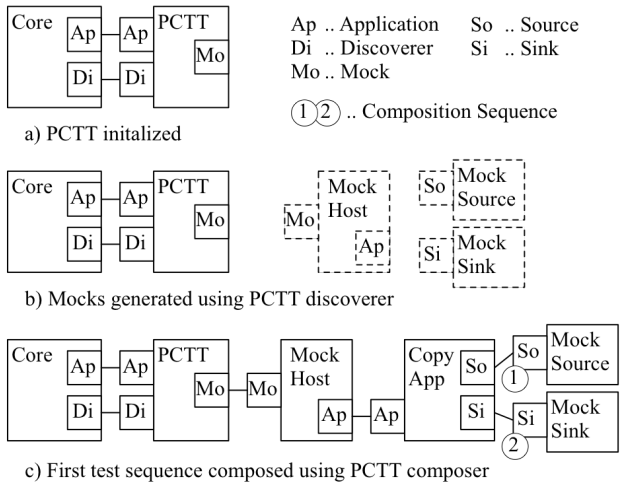


Fig. 13 PCTT integration in Plux, composition of test sequence.

V. Related Work

In component-based software development, the prevailing composition method is programmatic composition, where the host creates and integrates its contributors itself. Dynamic plug-and-play composition like in Plux is rather the exception. Common test methods are limited to functional unit tests of components and on system tests of the composed application, whereas composability is generally ignored.

A. Systems with Dynamic Reconfiguration

Component systems such as *OSGi* [5], *Eclipse* [6], and *NetBeans* [7] support dynamic reconfiguration. We looked at what they recommend for testing their components:

Although the *Eclipse* platform supports dynamic composition, this feature is rarely used. The Eclipse IDE itself does not make use of it, and so do most third-party plug-ins. Because dynamic composition is uncommon, composition testing is not considered by the suggested Eclipse test methods. Eclipse recommends *JUnit* [10] for functional testing and *SWTBot* [11] for user interface testing. In addition to that, the Eclipse *Test & Performance Tools Platform* Project [12] allows recording API calls for regression testing.

The *OSGi* Service Platform specification and the OSGi documentation [5] do not address testing at all. The *DA-Testing* project [13] recognizes the need for dynamic composition testing. It provides a framework which listens to the events of the OSGi service registry and runs unit tests when those events occur. The assertion API for functional testing is kept intentionally similar to JUnit.

In *NetBeans* [7], dynamic reconfiguration is considered in the API, but ignored by the majority of plug-ins. Thus applications built with NetBeans usually need to be restarted to add or remove plug-ins. The NetBeans project recommends testing the components with JUnit and provides a helper class to run unit tests inside the NetBeans environment. Composability testing is not considered.

In summary, the most modern component systems do not consider composability testing. This confirms the need for our work.

B. Systems without Dynamic Reconfiguration

For component systems without support for dynamic reconfiguration we looked at the inversion of control containers *Spring* [14] and *PicoContainer* [15]:

The *Spring* framework supports unit testing of applications. As Spring components are simple Java objects, unit tests can be conducted with test frameworks like *JUnit* [10] or *TestNG* [16]. To test classes that depend on external libraries or databases, Spring provides mock and utility classes. For enterprise applications, which require an application server, Spring supports integration testing. It allows executing the application in a spring environment and checking if the components are wired correctly, without the need to deploy the application to a server.

The *PicoContainer* project recommends the use of unit test frameworks like JUnit to test the components of an application, which are simple Java objects. To resolve dependencies between objects, PicoContainer recommends the use of mock libraries like *JMock* [17] and *EasyMock* [18].

We adopted the idea of mock objects for our work. Whereas Spring and PicoContainer use mock object for functional testing, we use mock components for composability testing.

VI. Conclusions

In this paper we presented the Plux Composability Test Method (PCTM) for unit testing of dynamically composed applications. The method determines and generates the test cases required to check the composability of plug-and-play components. In order to allow the independent testing of components, our approach generates a mock host, to host the component under test as well as mock contributors to test it. It enumerates all combinations of different contributors of a single host. In doing so, it generates a practically reasonable number of test cases.

We described the Plux Composability Test Tool (PCTT) and showed how it integrates into the Plux composition infrastructure. It generates test cases according to the PCTM and executes them.

By applying the method and the tool to our motivating example, we showed that the generated test cases effectively revealed composability defects. The tool allows time-efficient composability testing without programming, because it generates and executes the set of test cases automatically.

In future work, we will extend PCTM to test additional composability aspects: (i) We want to test hosts with multiple contributors of the same kind; both for their correct integration and for their correct use. (ii) We want to test if hosts behave correctly after contributors were removed. (iii) For contributors with multiple plugs, we want to test if their plugs can be connected individually and in arbitrary order. (iv) We want to test the composability of partially connected components. Such a component decides which contributors it uses depending on which of its provided plugs are connected.

Acknowledgment

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

References

[1] R. Wolfinger, "Dynamic Application Composition with Plux.NET: Composition Model, Composition Infrastructure.", Dissertation, Johannes Kepler University, Linz, Austria, 2010.

[2] C. Mittermair, "Zerlegung eines monolithischen Softwaresystems in ein Plug-In-basiertes Komponentensystem.", Master thesis, Johannes Kepler University, Linz, Austria, March 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] D. Birsan, "On Plug-ins and Extensible Architectures.", ACM Queue, 3(2):40–46, 2005.

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

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

[7] T. Boudreau, J. Tulach, G. Wielenga, "Rich Client Programming, Plugging into the NetBeans Platform", Prentice Hall International, 2007.

[8] "ECMA International Standard ECMA-335, Common Language Infrastructure (CLI)", 4th Edition, June 2006.

[9] M. Jahn, M. Löberbauer, R. Wolfinger, H. Mössenböck, "Rule-based Composition Behaviors in Dynamic Plug-in Systems.", Submitted to The 17th Asia-Pacific Software Engineering Conference, APSEC 2010, Sydney, Australia, November 30-December 3, 2010.

[10] K. Beck, E. Gamma, "JUnit Documentation", http://www.junit.org, retrieved June 2010.

[11] "The SWTBot Project", http://www.eclipse.org/swtbot, retrieved June 2010.

[12] "Eclipse Test & Performance Tools Platform Project", http://www.eclipse.org/tptp, June 2010.

[13] DynamicJava.org, "DA-Testing Project", April 2009.

[14] "Spring Java Application Framework, Release 3.0. Reference Documentation", http://www.springsource.org, June 2010.

[15] "PicoContainer", http://www.picocontainer.org, February 2010.

[16] C. Beust, H. Suleiman, "Next Generation Java Testing: TestNG and Advanced Concepts", Addison-Wesley Professional, October 2007.

[17] "JMock Project", http://www.jmock.org, retrieved June 2010.

[18] "EasyMock Project", http://easymock.org, retrieved June 2010.