JOHANNES KEPLER
UNIVERSITÄT LINZ | JKU

Technisch-Naturwissenschaftliche
Fakultät

Testing and Debugging of Dynamically Composed Applications

DISSERTATION

zur Erlangung des akademischen Grades

Doktor

im Doktoratsstudium der

TECHNISCHEN WISSENSCHAFTEN

Eingereicht von:

Markus Löberbauer

Angefertigt am:

Institut für Systemsoftware

Beurteilung:

o. Univ.-Prof. Dr. Dr. h.c. Hanspeter Mössenböck

a. Univ.-Prof. Dr. Johannes Sametinger

Mitwirkung:

Dr. Reinhard Wolfinger

Linz, Oktober 2012

Sworn Declaration

I hereby declare under oath that the submitted thesis has been written solely by me without any outside assistance, information other than provided sources or aids have not been used and those used have been fully documented.

The thesis here present is identical to the electronically transmitted text document.

Linz, October 2012 Markus Löberbauer

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Dissertation selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Die vorliegende Dissertation ist mit dem elektronisch übermittelten Textdokument identisch.

Linz, Oktober 2012 Markus Löberbauer

Abstract

Dynamic software composition allows users to reconfigure a program on-the-fly by adding, removing, or swapping sets of components. This requires that every component is dynamically composable, i.e., that it can extend other components and can be extended by other components at run time. To ensure that a component is composable in all possible composition scenarios, testing is necessary, and if an error shows up in a program, developers need debugging support to identify the causing component as well as information about what caused the error.

Current component systems only provide a test harness to host the component under test and recommend test methods, which are usually limited to unit tests of the business logic, whereas the composability of a component is neglected. This is insufficient for dynamically composed programs where every component must work in various composition scenarios.

In this thesis, we present a classification of the composition mechanisms used in current component systems and their contributor provision characteristics (i.e., in what way contributor components are provided to host components). Based on this classification, we present a classification of composability faults that are typical for the composition mechanisms. To find such faults in components, we present the composability test method *Act* for the Plux composition infrastructure. Act generates test cases for all composition scenarios by varying the order of composition operations, selects a representative subset of test cases so that they are executable in reasonable time, and executes the test cases. Executing a test case means composing the component under test using a given testbed, executing the functional tests, and collecting the occurring composition errors and functional errors. Furthermore, we present the test tool *Actor* which automates this test method. To find the causes of composability errors in components, we present the debugging method *Doc* for Plux. Doc records the composition operations of running Plux programs into composition traces, filters the composition traces for relevant components, splits the composition traces into parts with related composition operations, and allows

developers to compare traces in order to locate the cause of errors. Moreover, Doc creates hints for possible causes of errors using reasoning and allows replaying the composition from composition traces to visualize the composition state of the program.

This work was funded by the Christian Doppler Research Association and BMD Systemhaus GmbH.

Kurzfassung

Mit dynamischer Komposition können Benutzer ein Programm durch Austausch von Komponenten umkonfigurieren während es läuft. Dazu muss jede Komponente dynamisch komponierbar sein, d.h. sie muss damit umgehen können, dass die Komponente selbst oder andere Komponenten zur Laufzeit hinzugefügt oder entfernt werden. Um sicherzustellen, dass eine Komponente in allen möglichen Kompositionsszenarios komponierbar ist, muss getestet werden. Wenn ein Fehler gefunden wird, soll der Entwickler beim Identifizieren der fehlerhaften Komponente und beim Finden der Fehlerursache unterstützt werden.

Aktuelle Komponentensysteme bieten lediglich eine Testumgebung um die getestete Komponente zu komponieren und empfehlen eine Testmethode, die allerdings nur die Geschäftslogik umfasst, aber die Komponierbarkeit vernachlässigt. Für dynamische Komposition ist das zu wenig, weil jede Komponente in unterschiedlichen Kompositionsszenarios funktionieren muss.

In dieser Arbeit zeigen wir eine Klassifikation von Kompositionsmechanismen und deren Eigenschaften bei der Komponentenbereitstellung. Basierend darauf, zeigen wir eine Klassifikation von typischen Komponierbarkeitsdefekten für die gezeigten Kompositionsmechanismen. Um solche Defekte zu finden, zeigen wir die Testmethode *Act* für die Plux-Kompositionsinfrastruktur. Act erzeugt Testfälle für alle Kompositionsszenarios durch Variieren der Kompositionsreihenfolge, es wählt eine repräsentative Teilmenge dieser Testfälle, damit diese in vernünftiger Zeit ausführbar sind, und führt sie aus. Einen Testfall auszuführen bedeutet, dass die zu testende Komponente in einer gegebenen Testumgebung komponiert wird, funktionale Tests ausgeführt werden, und die auftretenden Kompositionsfehler sowie die funktionalen Fehler protokolliert werden. Weiters zeigen wir das Testwerkzeug *Actor*, das die Testmethode automatisiert. Um die Ursache von Komponierbarkeitsfehlern zu lokalisieren, zeigen wir die Debuggingmethode *Doc* für Plux. Doc zeichnet die Kompositionsoperationen eines laufenden Plux-Programms in einem Protokoll auf, filtert dieses nach relevanten Komponenten, teilt es in Teile zusammengehörender Kom-

positionsoperationen und ermöglicht Entwicklern diese Protokollteile zu vergleichen und damit die Fehlerursache einzugrenzen. Außerdem erzeugt Doc Hinweise auf mögliche Fehlerursachen und ermöglicht das schrittweise Abspielen von Kompositionsprotokollen, um den Kompositionszustand des Programms zu visualisieren.

Diese Arbeit wurde von der Christian Doppler Forschungsgesellschaft und von BMD Systemhaus GmbH gefördert.

Table of contents

Chapter 1: Introduction

Dynamic composition allows users to reconfigure a program on the fly by adding, removing, or swapping sets of components. Thus users can align a feature-rich program with the working situation at hand. With plug-and-play components, this can be done without configuration or programming effort. This flexibility requires that every component is dynamically composable, i.e., that it can extend other components and can be extended by other components at run time. To ensure that a component is dynamically composable, additional testing is necessary. Furthermore, if a program shows an error during dynamic composition, developers need debugging support to identify the causing component as well as information about what caused the error.

In practice, developers use unit tests to test the functionality of isolated components and they use integration tests to test whole programs. As these techniques neglect composability, they are insufficient for dynamically composed programs. A systematic composability test method is necessary. The number of variants in which plug-and-play components can be assembled into a program is enormous. Because testing all these variants manually is impractical, a tool is necessary to automate this.

This thesis presents a method and a tool for composability testing. The method shows how to generate test cases and how to reduce the vast number of generated test cases to a reasonable number that still finds most errors. The test tool applies the test method by automatically generating test cases and applying them to the components under test.

This thesis also presents a method and a tool for post-mortem composition debugging. The method shows how to locate composability faults in dynamically composed programs by analyzing recorded composition operations. The debugging tool allows developers to record a program execution, replay the composition events later, and search for the causes of composition errors.

Section 1.1 presents the context of dynamically composed programs and relevant questions that arise when such programs must be tested. Section 1.2 shows why composability testing and composition debugging are necessary. Section 1.3 summarizes the research contributions of this thesis. Section 1.4 gives an

overview of the project history and introduces the members of the Plux team with their contributions. Section 1.5 outlines the remainder of this thesis.

1.1 Research context

This thesis was conducted as part of the industrial research project: *Component architectures for next-generation business computing systems*. The goal of this project is to design and implement a component model and a composition infrastructure for extensible and customizable enterprise software. The project is conducted in cooperation between the BMD Systemhaus GmbH and the Christian Doppler Laboratory for Automated Software Engineering associated with the Institute of System Software at the Johannes Kepler University Linz. BMD builds enterprise resource planning software for small and medium-sized companies in Austria, Germany, Hungary, Italy, Switzerland, Slovenia, Croatia, Poland, Slovakia, and the Czech Republic.

In this project, we developed the Plux composition infrastructure [Wolfinger, 2010] as the basis for the next-generation business application of our industrial partner BMD. Programs developed with Plux comprise fine-grained components, which are assembled by the composition infrastructure using a plug-and-play approach. Users can adapt the program on the fly to align it with the working situation at hand by swapping sets of components.

A Plux program is open and extensible, i.e., a seamless program can be built from components of various manufacturers. To enable this, the manufacturers must specify contracts, publish them, and all components must strictly adhere to them. In dynamically composed programs, the composition is delayed from compile time to run time. At compile time, manufacturers must not make any assumption besides what is specified in the contracts, because the assumption is unlikely to hold when the component is used in another environment. This leaves manufacturers with the following uncertainties:

- It is unknown which other components will be available at run time. Thus all components must be self-contained and functional, independent of which other components are available. Components must not depend on the existence of other components.
- It is unknown by which other components a component will be used. As a contract defines the usage relation only in one direction, it is undefined which services the using component provides. Thus, a used component must not rely on its using component.
- During its lifetime, a component may be used by different other components. It may be disconnected from one component and reconnected to another com-

ponent, and it may even be connected to multiple components concurrently. Thus a component must not rely on a specific usage scenario.

- The order in which components are connected can vary, i.e., if a component uses multiple components (either with different contracts or with the same contract) the order in which the components are connected is undefined. Thus, if a component can use a component A only if a component B is available, it must be able to handle the case that A is connected before B.
- The order in which components are disconnected can vary, i.e., if a component uses multiple components (either with different contracts or with the same contract) the order in which the components are disconnected is undefined. Thus, if a component can use component A only if component B is available, it must be able to handle the case that B is disconnected before A.

As these uncertainties cannot be resolved at development time, the component manufacturer must consider all possible scenarios. To ensure the functionality of a component in all scenarios, the manufacturer must do extensive testing and debugging:

- Composability testing should perform functional tests of components within generated composition scenarios. It tests if a component works independently from other components it uses or to which it provides, and independently from the order in which components are connected and disconnected.
- Composition debugging should help identifying the causes of the errors detected during testing. It reveals the common characteristics of the composition scenarios in which errors occur.

The next section discusses the problems that must be solved to support developers of dynamically composable components with composability testing and composition debugging.

1.2 Problem statement

This thesis discusses the problem of how to test and debug dynamically composable components. Section 1.1 discussed which uncertainties and challenges the manufacturers of dynamically composable components face. How such components can be tested and debugged is an open research problem:

- Composability testing involves that a component is tested in all composition scenarios that can occur. The open research question is *how can a composition scenario be characterized and how can we determine relevant composition scenarios?*

- The number of relevant composition scenarios for composability testing is usually vast. The open research question is *how can we determine a subset of relevant composition scenarios that can be efficiently executed and still effectively reveals composability errors?*
- A test suite which tests a faulty component with different composition scenarios will produce a result which segments the composition scenarios into two groups: one that shows the error and another which does not. The open research question is *how can we determine the significant differences between those groups and thus the possible cause for the error?*

The lack of systematic testing and debugging limits dynamically composed programs to academia, because the stability is not sufficient for its application in industry. If the research questions above can be answered, developers can test the composability of their components systematically, similar to the way functional class tests are done today. By this, the promise made by dynamic composition infrastructures to enable developers to build extensible, customizable, and dynamically reconfigurable programs can be fulfilled also in industrial applications, because systematic testing ensures the stability that is required there.

1.3 Research contributions

We claim the following contributions in this thesis: a classification of composition mechanisms, a classification of composability faults, a method for composability testing, and a method for post-mortem composition debugging. Furthermore, we show the feasibility of the test method and the debugging method with tool implementations for the Plux composition infrastructure.

- *Classification of composition mechanisms*. We examined the composition characteristics of current component systems, deduced component mechanism classes, and ordered these classes by their support for adaptable programs. We use this classification to determine which composability faults can occur if a specific composition mechanism used. In general, this classification can be used to specify the adaptability in software requirements.
- *Classification of composability faults.* We identified the composability faults that components can have if a specific composition mechanism is used. We use this classification to name the appropriate test methods for each composition mechanism, to reveal possible composability faults in a component.
- *Composability test method.* We designed a method for composability testing. It generates an adjustable number of test cases, with a trade-off between run time and effectiveness: more test cases find more errors, but take longer to ex-

ecute. However, even for small numbers of test cases, the test method is effective, because it selects test cases that are likely to reveal errors.

- *Composition debugging method.* We designed a method for post-mortem composition debugging. It enables developers to locate composability faults, by analyzing recorded composition operations. The method can record individual program executions. During this process, it filters the composition operations that are likely to cause errors and marks the working and the failing operation sequences. The method can also record a series of test cases executions, which typically leads to vast amounts of recorded data. In order to analyze the recordings in a feasible time, the method reduces the amount of data, by grouping recordings where the same error occurred and by selecting representative recordings from each group.

Chapter 3 presents the composition mechanism classification and the composability fault classification. Chapter 5 presents the composability test method. Chapter 6 presents the composition debugging method.

1.4 Project history

Plux is a research project conducted by the Christian Doppler Laboratory for Automated Software Engineering associated with the Institute for System Software at the Johannes Kepler University Linz, in cooperation with the industry partner BMD Systemhaus GmbH. Plux comprises an infrastructure for dynamically composed desktop and web applications, as well as an infrastructure to test dynamically composable components.

At the time of this writing the Plux team comprises: the project manager Reinhard Wolfinger; the Ph.D. student Markus Jahn, who works on dynamically composable web applications; the Ph.D. student Markus Löberbauer and his master student Philipp Lengauer, who work on testing and debugging; and the master student Thomas Hribernig, who works on retrofitted security.

Our industry partner BMD Systemhaus initiated the project in an effort to build the basis for their next-generation enterprise application. The new application should be extensible with third-party plugins and reconfigurable at run time.

In 2006 we designed a component model based on the metaphor of slots and extensions [Wolfinger et al., 2006]. In 2007 we published a composition infrastructure and demonstrated novel applications, which can be reconfigured in a plug-and-play manner. For the first time, users could add components to a program and remove components from a program to adapt it to their working situation at hand, without programming, configuring, or even restarting the program. A further novelty was a visualizer that instantly showed the program's architecture and its

changes. Furthermore, we published integration models for the secure integration of untrusted third-party plugins [Wolfinger and Prähofer, 2007].

From 2008 to 2010 Plux was redesigned to reduce the programming effort for component developers. This has been accomplished with a richer composition model [Jahn et al., 2011], composition behaviors [Jahn et al., 2010a], and component templates [Wolfinger et al., 2010]. With the richer composition model, components can share information in a standardized manner; with behaviors, composition logic can be reused to control complex composition scenarios declaratively; and with component templates, custom components can be generated from generic component templates.

From 2010 to 2012 Plux was extended to support distributed multi-user web applications [Jahn et al., 2010b; Jahn et al., 2011]. We designed the method for composability testing [Löberbauer et al., 2010] and composition debugging, and implemented the corresponding tools [Lengauer, 2012]. We also created a model for retrofitting security in component-based programs [Wolfinger et al., 2012] and a security manager implementation for Plux [Hribernig, 2012].

During the whole project the following student projects were conducted based on Plux: Stephan Reiter and Christian Mittermair componentized a customer relationship management application [Reiter and Wolfinger, 2007], [Mittermair, 2009], Markus Jahn created a cross compiler infrastructure [Jahn, 2008], Mario Eder created a web site monitor [Eder, 2008], Rainer Pichler created a tool to record runtime statistics [Pichler, 2009], Zoltan Toth created a script interpreter for composition scripts, Andreas Gruber created a graphical composition tool for Plux programs [Gruber, 2010], Sabine Weiss created a highly extensible customer relationship management application [Weiss, 2010], Thomas Hribernig created a security add-on for license enforcement and retrofitted security [Hribernig, 2012], Philipp Lengauer created a composition debugger [Lengauer, 2012], Patrick Hagmüller ported the core elements of Plux from C# to Delphi [Hagmüller, 2012], Bernhard Schenkermayr created a highly customizable calculator [Schenkermayr, 2012], Thomas Reinthaler created an application builder [Reinthaler, 2012].

1.5 Structure of the thesis

This thesis is organized as follows: Chapter 2 discusses the state of the art testing approaches for component platforms. A historical overview shows how testing evolved in software engineering, and how componentization affected testing. A section defines the component terminology and another section the testing terminology used in this thesis. The final section introduces the testing approaches

of current component platforms and analyzes the deficiencies of these approaches.

Chapter 3 analyzes and classifies the composability faults that can occur in component-based programs. It describes the different component provision characteristics of component platforms, defines composition mechanisms, and classifies the component platforms by their composition mechanism. Finally, the chapter describes and classifies the composability faults that can occur depending on the used composition mechanism.

Chapter 4 introduces the Plux composition infrastructure and shows how Plux assembles component-based programs. It describes how components specify their provided and required services using metadata, how Plux uses these metadata to connect required and provided services, and how Plux stores these connections in the composition state. A further section describes the architecture of the Plux composition infrastructure, and a concluding section derives the composition standards that are required by Plux in order to enable dynamic composition.

Chapter 5 describes how the composability faults from Chapter 3 can be addressed with composability testing. It describes a composability test method and how this method can be applied to find errors in Plux components. A further section describes a composability testing tool, which automates the testing method. The chapter concludes with the results of an experimental evaluation of the tool.

Chapter 6 describes how the cause of composition errors can be located with composition debugging. It describes a composition debugging method and how this method can be applied to locate the causes of composition errors on Plux programs. The chapter concludes with a description of a composition debugger, which supports the application of the debugging method.

Chapter 7 summarizes the contributions of this thesis, concludes how the contributions address the problem statement, presents ideas for future research, and closes with an overview of the current state.

Chapter 2: State of the art

According to G. J. Myers [1979], software testing is a process, designed to ensure computer code does what it is designed to do and that it does not do anything unintended. This process, though hard, is well researched for functional testing of statically linked programs. In dynamically composed programs however, the process is even harder. The behavior of a component can change at run time depending on the current composition, i.e., when other components are added or removed while the program is running. This chapter shows that the support for testing the composability of components is inadequate in current testing methods and that further research is necessary.

This chapter is structured as follows: Section 2.1 gives a historical overview on how testing evolved in software engineering and how componentization affected testing. Section 2.2 defines the component terminology, and Section 2.3 the testing terminology relevant for this thesis. Section 2.4 analyzes the testing approaches of current component platforms and analyzes their deficiencies.

2.1 Historical overview

Gelperin and Hetzel [1988] divide the history of software testing into the following periods. Until 1956, testing focused on finding faults in the hardware (debugging-oriented period). This period had no systematic approach to detect faults in the software.

In 1957, Baker distinguished debugging from testing. Debugging makes sure that a program runs, whereas testing makes sure that a program solves the problem. According to Baker, the goal was to demonstrate that a program has not faults (demonstration-oriented period). Both debugging and testing meant efforts to detect, locate, identify, and correct faults. These two activities were distinguished based on their definition of success: debugging succeeds if a program runs; testing succeeds if a program solves the problem [Baker, 1957].

In 1979, the destruction model redefined the meaning of debugging and testing. Testing includes all efforts to detect a fault, whereas debugging includes all efforts to locate, identify and correct a fault. Myers described the destruction testing as "the process of executing a program with the intent of finding errors". This defini-

tion shifts fault detection in the focus of testing. Myers argued that destructive tests are more likely to find bugs [Myers, 1979]. Furthermore, Myers associated testing with other fault detection activities such as analysis and review techniques.

In 1983, a standardized guideline describes how to use software testing to evaluate the quality of software during the software life-cycle (evaluation-oriented period). For each phase in the software life-cycle, a specifically chosen set of techniques (e.g., testing, analysis, or review) ensures that the development and maintenance of software meets specified quality requirements [NBS, 1984].

Since 1985, testing is planned early in the development cycle and performed in parallel with development. Programmers are provided with the test plan before they start the development (prevention-oriented period). Test cases are kept, maintained, and performed repeatedly in the software life-cycle to prevent regression [ANSI, 1987].

After the observation of Gelperin and Hetzel further progress was made in the field of software testing. In 1999, Beck describes test-driven development as part of Extreme Programming. Test-driven development is a software development process with short development cycles designed around automated test cases. Before a developer implements a new function, he writes a test case for this function. This test case will fail, because the function is not yet implemented. To implement the function, he writes code for the function, repeatedly reruns all test cases and corrects his code until all test cases pass. The fact that automated tests are available at any time, encourages developers to refactor their code, and allows them to add new features without having to be afraid of breaking existing features [Beck, 1999].

2.2 Component terminology

According to Heineman and Councill [2001], a component is a software element that conforms to a component model. A component model defines standards for composition and interaction. *Composition* is the process of combining components to yield new behavior. The *composition standard* specifies the rules how composition must be done, e.g., how a component can be replaced by another component. Interaction is the communication between components to realize this behavior. Components can provide functionality as well as request it. The *interaction standard* specifies how components must declare their provisions and requests using interfaces, so that other components can interact with them. An interface is an abstraction that describes the behavior of a component. A component can support multiple interfaces. Components, which adhere to the same *component model*, i.e., they respect its composition and interaction

standard, can be reused (without modification) to create new behavior by composition. To execute a component-based program a component model implementation is necessary. A component model implementation comprises software elements that support the execution of components that conform to the component model. A *software component infrastructure* is a set of interacting software components designed to ensure that a software system constructed using these components will satisfy defined performance specifications.

According to Weinreich and Sametinger [2001], a component model defines standards for contracts, naming, metadata, interoperability, customization, composition, evolution, as well as packaging and deployment. In this thesis, the following standards are significant: composition (terminology already defined above), interfaces, metadata, packaging and deployment.

- *Interfaces* are specifications of component behavior. The purpose of using component technology is black-box reuse: components reveal as little as possible about their inner mode of operation and clients of a component rely only on its interfaces. An interface serves as the contract between a component and its clients. By using interfaces, components may be modified or replaced, as long as they still fulfill their contract. The interface standard in a component model specifies how operations are specified. An operation is described by its name and parameters.

- *Metadata* are used to specify information about interfaces, components, and their relationships. Composition tools can retrieve the metadata to combine components by matching requested and provided interfaces. The metadata standard in a component model specifies how metadata are described and how they can be retrieved. Component model implementations must provide a dedicated service that allows retrieving metadata.

- *Packaging* is the means to assemble components that can be independently installed and configured. Therefore a component must include its implementation and all resources needed for operation. *Deployment* means to install and configure such a component in a component infrastructure. The packaging and deployment standard in a component model specifies how components are packaged and deployed.

Wolfinger [2010] adds the terms extension, host, and contributor. He also extends the component model with a composition state, defines the composition infrastructure as a subset of the component infrastructure, and distinguishes between programmatic composition and automatic composition.

- An *extension* is a component; it can be in the role of a host and a contributor. A *host* is an extension that requests the service of other extensions, and a *contributor* is an extension that provides such a service. The name extension goes back to the Gamma's extension object pattern, which can be used to add new interfaces to existing classes without modifying them by adding extension objects [Gamma, 1996]. Referring to this pattern, Wolfinger calls components extensions to emphasize the fact that a contributor extends a host with additional features (note: the host is functional without the contributor as well). Extensions can be in the role of a host and a contributor at the same time.

- The *composition state* is an addition to the component model. It comprises the instantiated extensions and their usage and provision relations. Accordingly, the implementation of a component model must have a service to maintain the composition state. This service can be used, for example, by the extensions to retrieve their contributors, or by tools to save and restore program snapshots.

- The *composition model* is the part of the component model that is responsible for discovering extensions, composing them, and maintaining the composition state. *Discovery* means to detect extensions and extract their metadata. Wolfinger [2010] recognized that existing component models lack support for automatic composition and a composition state, which leads to an unnecessarily high programming effort for component developers. He proposed to substitute the composition support in existing component models by extending them with a more capable composition model. A composition infrastructure is the implementation of a composition model.

- Composition can be classified into programmatic composition and automatic composition. *Programmatic composition* means that the host has to query a component registry has to discover, create and connect its contributors itself. *Automatic composition* means that components just declare their requests and provisions using metadata. The composition infrastructure uses these metadata to match requests and provisions and to discover, create, and connect matching components automatically.

This thesis uses the terminology from Sametinger and Weinreich with the additions of Wolfinger. However, it should not go unmentioned that Szyperski [2002] uses a different terminology and defines the following key terms:

- A *component system architecture* is a platform with a set of component frameworks and an interoperation design for the component frameworks. The *platform* is the base that allows installing components and component frameworks, such that these can be instantiated and activated.

- A *component framework* implements protocols to connect components and enforces policies defined by the component framework.
- An *interoperation design* for component frameworks comprises the rules of interoperation among all the frameworks joined by the component system architecture.
- A *component* is a set of simultaneously deployed atomic components.
- An *atomic component* is a module, i.e., a set of classes and/or non-object-oriented constructs, such as procedures or functions, and a set of resources.

2.3 Testing terminology

According to Beizer [1990] an *error* is an incorrect behavior (symptom) resulting from a fault. A *fault* is an incorrect program or data object (bug). He distinguishes two categories of fault detection methods: *static analysis*, which is done on the source code without executing the program; and *dynamic analysis*, which is done by executing the program and checking calculated values against expected values.

According to Myers [1979] testing is the process of executing a program with the intent of finding errors. As such it is a destructive process and thus most people find it difficult. Therefore, a successful test is one that detects an undiscovered error. Debugging is the process of determining the reason of the error in the program as well as correcting the fault. Changes in the program can introduce new faults. To detect such faults, *regression testing* is necessary. Regression testing means to keep the test cases and to execute them after changes.

Myers distinguishes the testing strategies black-box and white-box testing. *Black-box testing* means testing a program without any knowledge about its internal structure. It is input/output-driven. The tester derives the test input data from the program's specification. He tries to find input data for which the program does not calculate the correct output data according to the program's specification. In contrast, *white-box testing* means testing a program with knowledge about its internal structure. It is logic-driven. The tester examines the program's internal logic and chooses test input data, which satisfies a specified *coverage criterion*. Beizer [1990] distinguishes the following coverage criteria: *statement coverage* means that all statements of a program must be executed; *branch coverage* means that every branch alternative must be taken, and *path coverage* means that all possible control flow paths through the program must be executed.

Myers also distinguishes various test scopes, such as module tests and function tests. *Module tests* (*unit tests*) find errors in individual modules of a program. *Function tests* (*integration tests*) find errors in the whole program.

2.4 Testing approaches of component platforms

In practice, developers produce rather monolithic programs and their testing focuses on the program logic. Even if developers decide to use a componentized software architecture, the testing still focuses only on the program logic, and neglects the composability. When using a static composition mechanism (which is most common), the composition is the same for every execution of the program, so it is easy to test against this composition. With dynamic composition this is far more critical, because many errors occur only when the user reconfigures a program at run time. Furthermore, component-based programs are usually composed programmatically, which limits the kinds of possible faults, because the developer controls the composition process, i.e., the time and the order in which components are assembled. However, if programs are composed automatically, the composition infrastructure (and not the programmer) controls the composition process. Components must comply with this composition process, i.e., they must be composable. Thus their composability must be thoroughly tested.

We examined the testing approaches for dynamic composition as well as for static composition by means of typical representatives.

2.4.1 Component systems with static composition

We looked at the component platforms *Spring* [Johnson et al., 2011] and *PicoContainer* [PicoContainer, 2012] as representatives of systems without dynamic composition:

The Spring framework supports unit testing of programs [Johnson et al., 2011]. As Spring components are plain Java objects, unit tests can be conducted with test frameworks like *JUnit* [2011] or *TestNG* [Beust and Suleiman, 2007]. 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 allows 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 to test the components of a program, which are plain Java objects. To resolve dependencies between objects, PicoContainer recommends the use of mock libraries like JMock [2012] and EasyMock [2012].

None of these component systems takes into account that components can be used in compositions other than those which the developer tested. There is no systematic approach to determine which compositions should be tested, thus errors that occur only in specific compositions are not detected.

2.4.2 Component systems with dynamic composition

Component systems such as *OSGi* [2011], *Eclipse* [2003], and *NetBeans* [Boudreau et al., 2007] 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 in practice. The Eclipse IDE itself does not make use of it, and so do most third-party plugins. Because dynamic composition is uncommon, the suggested Eclipse test methods do not care about composability testing. Eclipse recommends *JUnit* [2011] for functional testing and *SWTBot* [2012] for user interface testing. In addition to that, the Eclipse *Test & Performance Tools Platform* Project [2012] allows recording API calls for regression testing.

The OSGi Service Platform specification and the OSGi documentation [OSGi, 2011] do not address testing at all. The DA-Testing project [Abu-Eid, 2009] recognizes the need for dynamic 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. The DA-Testing project appears to have been discontinued in 2009 for reasons unknown to us.

In *NetBeans* [Boudreau et al., 2007], dynamic reconfiguration is considered in the API, but ignored by the majority of plugins. Thus programs built with NetBeans usually need to be restarted in order to add or remove plugins. The NetBeans project recommends testing the components with JUnit and provides helper classes to run unit tests inside the NetBeans environment.

Like component systems with static composition, none of the component systems with dynamic composition provides means for testing different compositions. Furthermore, they also do not take into account that the composition process is dynamic, thus errors that occur only in specific composition sequences are not detected. Even if a composability error is found accidentally, none of these component systems provides support for locating the cause of the error.

2.5 Component testing in literature

The need for component testing is widely recognized in the literature. Many papers give approaches to ensure that components work as expected when integrated into a program. We figure that the following papers represent the state of the art in component testing research.

Weyuker [1998] recognizes that testing a component in one project does not guarantee that the same component will work in another project. Especially because testing is generally done by developers with knowledge about the component under test and its use in the initial project. She thus suggests that components should be re-tested for each new project. Weyuker discusses the use of components in statically linked programs, while this thesis discusses component usage in dynamically reconfigurable programs. In dynamically composed programs every possible configuration can be seen as a new project, thus we conclude that an automated systematic test method is necessary.

Liu and Dasiewicz [1999] describe an approach for testing the interaction between components in a program that is based on formal models of the program's components. The approach can automatically create integration tests for a program. As limitations of this approach, Liu and Dasiewicz recognize the high costs for creating the formal models and the limited scalability of the used algorithms.

Gao [2000] defines the properties that a component must have in order to be testable: a component must be *observable* during usage (i.e., its input and output parameters must be visible), must be *trace-controllable* (i.e., provide functions to monitor and check the component's behavior), and must be *understandable* (i.e., the vendor must provide information about how the component must be used).

Wu et al. [2001, 2003] specify test criteria for component-based software based on a component interaction model. As component vendors usually do not provide an interaction specification or the source code, the papers suggest to model the desired interaction between the components in UML.

Kranzlmüller et al. [2002a, 2002b] describe an approach for testing nondeterministic parallel programs. The approach records the nondeterministic receives during a program execution, generates all possible receive orders, and replays the program execution with deterministic receives in all possible orders. As the limitation of the approach the authors recognize its scalability; the approach has so far been applied to programs with up to 8 processes.

Bertolino and Polini [2003] suggest that component developers should include test cases for the components they provide. By this, users can see what kind of testing a component already underwent and adapt their integration tests to it.

Furthermore, they present an approach to decouple integration tests from the components by using adapters. The goal of the approach is to allow early writing of test cases and using the test cases for alternative candidates of used components.

Mariani et al. [2005, 2007] present an approach to reduce the number of test cases which must be executed to ensure that a program still works after components-of-the-shelf are updated or replaced with other interface-compatible components. The approach is based on behavioral models that represent the component interactions. The models are generated while a test suite executes the program with the components that are subject to replacement. Later these models are used to select the test cases from the test suite that are relevant to test the program with the new components. The goal is the reduce the necessary number of test cases and thus the time needed to check if a new version of a component or another interface-compatible component integrates well into the program.

Zeng et al. [2007] recognize the need for re-testing parts of a program that are built with components-of-the-shelf, when new releases of these components become available. The paper presents an approach that allows selective re-testing by analyzing: the binaries of the old version and the new version of the components, the source code of the program, and the test suite of the program. The result is a reduced test suite that covers the changes in the components. This works with components written in C/C++ that are deployed in the common object file format or as portable executables.

Hewett and Kijsanayothin [2009] present an approach for automatically generating the orders in which components must be tested to minimize the number of required test stubs, based on a given component test dependency graph. This approach incrementally tests and integrates components, i.e., it uses already tested components as stubs to test other components and so forth.

Saglietti and Pinte [2010] recognize that integration tests must be performed every time a component is used in a new context. Their approach is based on communication state machines, which reflect the impact of component-internal values on external invocations. Based on this state machines, their approach generates test suites consisting of test cases. A test case is a set of message sequences that are sent to the interacting components, enabling the components to cover the required interactions.

Besson et al. [2011] suggest the test-driven development of web services and service oriented architectures (SOA). Web services as the smallest units of soft-

ware in SOA should be tested with unit tests, web service choreographies with integration tests.

Da Silva and de Lemos [2011] present an approach to generate test plans for integration tests of self-adaptive software. These test plans are used to test new components whether they operate as specified when they are integrated into the self-adaptive system. This approach requires the following artifacts: test cases for the components, mechanisms for calculating the integration order of the components, and all stubs necessary for testing.

The approaches proposed by these papers cover functional testing and integration testing of components, mostly for components that are integrated into a program at development time. However, they do not specifically cover the dynamic composability of components, as is required to test plug-in components that are added to and removed from programs at run time.

Chapter 3: Composability faults in component-based programs

In component-based programs contributor components provide services and host components use them. The means of how a host and a contributor are connected depend on the used composition mechanism. In this chapter, we define what a composition mechanism is, classify composition mechanisms by their contributor provision characteristics, and classify component systems by the composition mechanisms they offer. The provision characteristics of a certain composition mechanism correspond to a specific set of possible faults. As this is relevant for testing, we define which faults in hosts and contributors can occur if a certain composition mechanism is used. [Löberbauer et al., 2012]

3.1 Contributor provision characteristics

In a component system, contributor provision is the process that connects a contributor to a host. This process includes the following activities: the contributors make their services available, the hosts identify contributors matching their requested services, and the hosts instantiate their contributors. The means of how this process works depends on the used component system. Each component system has specific contributor provision characteristics: it can make the contributors available at different points in time; it can maintain or not maintain the requested contributors in a registry, and if so, it can maintain them globally or in a host-specific way, and with or without the requesting host's instance; and it can connect a single contributor or multiple contributors to a host.

From the activities in the contributor provision process and from the contributor provision characteristics of the component system we derive the following test-relevant *contributor provision aspects*:

Identification .. how does a host identify the contributors it desires?

Instantiation .. how does a host get an instance of a contributor?

Availability .. when is a contributor available?

Registration .. what does the component system store about a contributor?

Cardinality .. with which number of contributors must a host work?

The following sections explain the contributor provision aspects and their characteristics by the means of an example program. The program is a library, composed of a user interface host that lists books from contributors that store books. The book store contributors contain a set of books, which is persisted when a book store is deactivated and restored when activated. The library host retrieves the books from its book store contributors and presents the books in its user interface. Figure 1 shows the user interface of the library application and Figure 2 shows its components, i.e., the library host that shows the books stored in the local and the offsite book store.

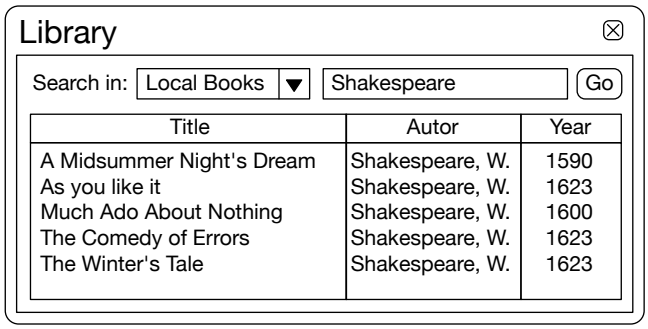


Figure 1: User interface of the library application

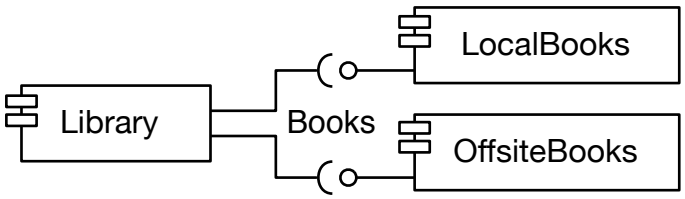


Figure 2: Components of the library application

3.1.1 Contributor identification

In order to request a contributor, a host must identify which contributor it wants to use. A host can do so, either by identifying a specific component as its contributor, or by specifying a contract for which an arbitrary contributor can provide an implementation. Let us look at a Java implementation for both scenarios.

3.1.1.1 By component

For this example we use Java classes to implement the components *Library* (cf. Figure 3), *LocalBooks*, and *OffsiteBooks*. In the library host, we create the *LocalBooks* and *OffsiteBooks* contributors with *new* and statically link them to *Library* at compile time.

```
class Library {
  LocalBooks local;
  OffsiteBooks offsite;
  Library() {
    local = new LocalBooks();
    offsite = new OffsiteBooks();
    // ... create the user interface ...
  }
}
```

Figure 3: Host that identifies its contributors by component

3.1.1.2 By contract

For the provision by contract implementation, we use the same contributor components as in Section 3.1.1.1. The components *LocalBooks* and *OffsiteBooks* (not shown) fulfill the book contract as specified by the *Books* interface (cf. Figure 4).

```
// File: Books.java
interface Books {
  // ... methods to access the book store ...
}

// File: META-INF/services/Books
LocalBooks

// File: LocalBooks.java
class LocalBooks implements Books {
  // ... methods to access the local book store ...
}
```

Figure 4: Contract and implementation of a contributor

In the library host, we use the *Books* interface to identify all contributors for the book contract. By using the *Java service loader* [Oracle, 2006] we iterate over the contributors, create them, and store them for later use. The contributors are dynamically looked up at startup time. As the number of available contributors can vary, we store them in dynamically growing list (cf. Figure 5). For this example to work, we compile the *Books* interface to the *Books.jar* contract file, and reference the contract when we compile the library, the local, and the offsite book contributor.

```
class Library {
  List<Books> bookStores;
  Library() {
    bookStores = new ArrayList<Books>();
    ServiceLoader<Books> bookServices =
        ServiceLoader.load(Books.class);
    for (Books b : bookServices) {
      bookStores.add(b);
    }
    // ... create the user interface ...
  }
  // ... main method, etc. ...
}
```

Figure 5: Host that identifies its contributors by contract

3.1.2 Contributor instantiation

In order to use a contributor, the host needs an instance of it. The host can create the instance itself or the composition infrastructure can provide the instance. The Java examples in Section 3.1.1 cover both scenarios: in Figure 3, the host creates the local and offsite book store with the *new* statement; whereas in Figure 5 the host retrieves the instantiated stores from the *Java service loader* [Oracle, 2006].

A composition infrastructure can create the contributor instances in a globally uniform way or in a host-specific way. Globally uniform means that the contributor instances for all hosts are provided in the same way, either one dedicated instance that is shared among all hosts, or separate instances for each host. Host-specific means that a shared instance is provided to some hosts, whereas separate instances are provided to other hosts, depending on the desired composition. The Java service loader example in Figure 5, Section 3.1.1 covers the globally uniform scenario. The service loader uniformly provides a new instance to every host that requests a contributor.

The *PicoContainer* [PicoContainer, 2012] example in Figure 6 covers the host-specific scenario. In this example, the contributors are injected into the constructors of the hosts. We use separate pico containers to control whether a shared contributor or unique contributors is injected. As the local library and the local web library (not shown) are in the same container, they get a shared local book store contributor. The other web library is in a separate pico container and thus gets a different instance of the local book store contributor.

```
class Application {
  // ...
  static void main(String[] args) {
    DefaultPicoContainer picoLocal = new DefaultPicoContainer();
    picoLocal.addComponent(Library.class);
    picoLocal.addComponent(WebLibrary.class);
    picoLocal.addComponent(LocalBooks.class);
    picoLocal.start();
    Library localLib =
        (Library) picoLocal.getComponent(Library.class);
    Library localWebLib =
        (Library) picoLocal.getComponent(WebLibrary.class);
    DefaultPicoContainer picoWeb = new DefaultPicoContainer();
    picoWeb.addComponent(WebLibrary.class);
    picoWeb.addComponent(LocalBooks.class);
    picoWeb.start();
    Library webLib =
        (Library) picoWeb.getComponent(WebLibrary.class);
    // ... use libraries ...
  }
}
class Library {
  List<Books> bookStores;
  Library(Books[] bookStores) {
    bookStores = Arrays.asList(bookStores);
    // ... create the user interface ...
  }
}
```

Figure 6: Application with host-specific contributor instantiation

3.1.3 Contributor availability

The contributors used by a host can become available at different times, either when the host is instantiated or later at run time. If a contributor becomes available at run time, the host can be notified by the composition infrastructure about the change, or needs to poll for changes otherwise. The contributors can be permanently available to the host, or only temporarily if the composition infrastructure allows removing them at run time.

The composition infrastructure provides the contributors to the host in a certain order. The contributors may be provided in a predictable order, i.e., in the same order for each program execution, or in an unpredictable order. The contributors may be provided all at once or continuously. The order in which contributors are provided matters — both for contributors for the same contracts and for contributors for different contracts.

Let us look at some examples.

3.1.3.1 At host instantiation time

To show how a host can retrieve its contributors at instantiation time, we use the library application in an implementation with constructor injection based on PicoContainer [PicoContainer, 2012]. In this solution, PicoContainer provides all contributing book stores to the library's constructor when the library is created (cf. Figure 7).

```
class Library {
  List<Books> bookStores;
  Library(Books[] bookStores) {
    bookStores = Arrays.asList(bookStores);
    // ... create the user interface ...
  }
  // ... main method, etc. ...
}
```

Figure 7: Host that retrieves its contributors at instantiation time

3.1.3.2 Later at run time

To integrate contributors later at run time, the library host needs to poll its contributors. We use the *Java service loader* [Oracle, 2006] to retrieve the available contributors. When the library application refreshes its user interface, it queries the service loader and updates the list of available contributors (cf. Figure 8). The service loader will provide new contributors as they become available at run time.

```
class Library {
  Set<Books> bookStores = new HashSet<Books>();
  ServiceLoader<Books> bookServices;

  Library() {
    bookServices = ServiceLoader.load(Books.class);
    // ... create the user interface ...
  }

  void updateBookStores() {
    bookServices.reload();
    for (Books b : bookServices) {
      bookStores.add(b);
    }
  }
  // ... main method, etc. ...
}
```

Figure 8: Host that retrieves its contributors later at run time

3.1.3.3 With notification

To integrate contributors later at run time a host can poll its contributors a shown in Section 3.1.3.2. However, if the composition infrastructure notifies the host when new contributors are available, the host can react to the changes immedi-

ately. To demonstrate this, we show an example using the *OSGi* service registry [OSGi, 2011]. The service registry maintains all available contributors. Using a tracker, a host can receive a notification (i.e., a call to its method *addingService*) when a new contributor is added to the service registry (cf. Figure 9; note, the code is simplified for the sake of shortness).

```
class Library {
  List<Books> bookStores;
  BooksTracker tracker;

  class BooksTracker extends ServiceTracker {
    BooksTracker(BundleContext context) {
      super(context, Books.class.getName(), null);
    }
    Object addingService(ServiceReference ref) {
      Books books = (Books) context.getService(ref);
      bookStores.add(books);
      return books;
    }
  }

  Library(BundleContext context) {
    bookStores = new CopyOnWriteArrayList<Books>();
    // register tracker for contributors
    tracker = new BooksTracker(context);
    tracker.open();
    // get current contributors
    for (ServiceReference ref : tracker.getServiceReferences()) {
      Books b = (Books) tracker.getService(ref);
      bookStores.add(b);
    }
    // ... create the user interface ...
  }
  // ... main method, etc. ...
}
```

Figure 9: Host that retrieves its contributors when it is notified

3.1.3.4 Permanently

A composition infrastructure can make the contributors to the hosts available permanently. This means that once a host has retrieved a contributor and stores a reference to it, the contributor cannot be removed from the host, i.e., the host can use it permanently. Figure 9 shows a host that uses its contributors permanently.

3.1.3.5 Temporarily

A composition infrastructure that supports run-time reconfiguration makes contributors only temporarily available to hosts. This means that the hosts are notified when contributors become available, as well as when contributors are removed. Figure 10 shows the host implementation from Figure 9 with an extended service

tracker that reacts to removal notifications by releasing the contributor via the method *ungetService*.

```
class Library {
  List<Books> bookStores;

  class BooksTracker extends ServiceTracker {
    // ... constructor and addingService shown in Figure 9
    void removedService(ServiceReference ref, Object service) {
      Books books = (Books) service;
      context.ungetService(ref);
      bookStores.remove(books);
    }
  }
  // ... constructor shown in Figure 9
}
```

Figure 10: Host that uses its contributors temporarily

3.1.3.6 In predictable order (same contract)

Contributors for a contract can be provided to the host in predictable order, i.e., the host gets the contributors in the same order on every program run. To demonstrate this, we refer to the example in Figure 3 where the host identifies the contributors by component and instantiates them using the *new* statement in the desired order:

```
local = new LocalBooks();
offsite = new OffsiteBooks();
```

3.1.3.7 In unpredictable order (same contract)

The order in which the composition infrastructure provides contributors to a host can be unpredictable. For example, a composition infrastructure with this behavior is the Java service loader. The order in which the service loader provides contributors depends on the class path. If the host retrieves the contributors as shown in Figure 11, the order of the contributors depends on the configured class path.

```
ServiceLoader<Books> bookServices = ServiceLoader.load(Books.class);
for (Books b : bookServices) {
  // ... use the books contributor ...
}
```

Figure 11: Host that retrieves its contributors in unpredictable order

3.1.3.8 All at once (same contract)

The host can retrieve all contributors for a contract at once. An example for this is the constructor injection as used in *PicoContainer* [PicoContainer, 2011]. The host shown in Figure 12 gets all contributors in an array, which is passed to its constructor when it is created.

```
class Library {
  List<Books> bookStores;
  Library(Books[] bookStores) {
    bookStores = Arrays.asList(bookStores);
    // ... create the user interface ...
  }
  // ... main method, etc. ...
}
```

Figure 12: Host that retrieves all its contributors at once

3.1.3.9 Continuously (same contract)

A composition infrastructure that supports dynamic discovery can provide the contributors to the host continuously as soon as they become available. In Figure 13, the *Plux* [Wolfinger, 2010] composition infrastructure sends a notification to the host for each available contributor, i.e., it calls the method *AddBookStore* for every *Books* contributor that becomes available.

```
[Extension]
[Slot("Books", OnPlugged="AddBookStore")]
class Library {
  List<Books> bookStores = new List<Books>();
  void AddBookStore(CompositionEventArgs args) {
    bookStores.Add((Books) args.Plug.Extension.Object);
  }
}
```

Figure 13: Host that receives its contributors continuously

3.1.3.10 In predictable order (different contracts)

A host that uses multiple contracts can retrieve the contributors for these contracts in predictable order, i.e., it retrieves all contributors for one contract before it retrieves all contributors for the next contract, and so forth. To demonstrate this behavior, we extend our library example (cf. Figure 2 on page 19) with a logger contract. The library uses the logger to print the name of each book store contributor when it is added. Creating the contributors for the different contracts in predictable order, i.e., creating the logger with the *new* statement before calling the *add* method, ensures that the logger is ready when needed (cf. Figure 14).

```
class Library {
  List<Books> bookStores = new ArrayList<Books>();
  Logger logger;

  Library() {
    logger = new Logger();
    add(new LocalBooks());
    add(new OffsiteBooks());
  }

  void add(Books b) {
    bookStores.add(b);
    logger.print("Book store added: " + b.getName());
  }

  // ... main method, etc. ...
}
```

Figure 14: Host that retrieves its contributors for different contracts in predictable order

3.1.3.11 In unpredictable order (different contracts)

A composition infrastructure that supports dynamic discovery can provide the contributors for different contracts in unpredictable order. In Figure 15, the *Plux* [Wolfinger, 2010] composition infrastructure sends notifications when a *Books* contributor or a *Logger* contributor is available. The order in which Plux provides the contributors for these contracts is undefined, i.e., the book store could be provided before the logger is ready. The method *AddBookStore* cannot rely on an initialized logger and must therefore use an *if*-statement to avoid potential errors.

```
[Extension]
[Slot("Books", OnPlugged="AddBookStore")]
[Slot("Logger", OnPlugged="SetLogger")]
class Library {
  List<Books> bookStores = new List<Books>();
  Logger logger;

  void SetLogger(CompositionEventArgs args) {
    logger = (Logger) args.Plug.Extension.Object;
  }

  void AddBookStore(CompositionEventArgs args) {
    Books books = (Books) args.Plug.Extension.Object;
    if (logger != null) {
      logger.print("Book store added: " + b.getName());
    }
    bookStores.Add(books);
  }
}
```

Figure 15: Host that retrieves its contributors for different contracts in unpredictable order

3.1.3.12 All at once (different contracts)

A host that uses multiple contracts can retrieve the contributors for all contracts at once. Figure 16 uses constructor injection based on *PicoContainer* [PicoContainer, 2011]. The constructor of the library gets all contributors passed as arguments at once, i.e., it gets the contributor for the *Logger* contract, as well as the contributors for the *Books* contract.

```
class Library {
  Logger logger;
  List<Books> bookStores;
  Library(Logger logger, Books[] bookStores) {
    this.logger = logger;
    bookStores = Arrays.asList(bookStores);
    // ... create the user interface ...
  }
  // ... main method, etc. ...
}
```

Figure 16: Host that retrieves contributors for all contracts at once

3.1.3.13 Continuously (different contracts)

A composition infrastructure that supports dynamic discovery can provide the contributors for different contracts continuously, i.e., a contributor for one contract can be followed by a contributor for another contract, again followed by a contributor for the first contract, and so on. In the Plux-based host implementation shown in Figure 15, Plux provides the contributors not only in unpredictable order, but also continuously, i.e., a book store can be followed by a logger, which is again followed by further book stores.

3.1.4 Contributor registration

A composition infrastructure can store in a registry which contributors are available, which of them have already been provided to hosts, and which are currently in use. Such a registry can be used by the hosts of an application to retrieve contributors or by tools to keep track, which contributors are in use by which hosts. We classify composition infrastructures into those, which maintain the registry globally and those which maintain it in a host-specific way.

3.1.4.1 Global availability

In order to provide a contributor to a host, the composition infrastructure must know which contributors are available. To do so, it can store a global set of components commonly available to every host. Whenever a host requests a contributor, the composition infrastructure provides it. For example the Java service loader maintains such a global registry. Figure 17 shows how hosts can query the

service loader for contributors. As the registry is global, every host retrieves the same set of contributors.

```
ServiceLoader<Books> bookServices =
    ServiceLoader.load(Books.class);
for (Books b : bookServices) {
  // ... use the books contributor ...
}
```

Figure 17: Host that retrieves its contributors from a global registry

3.1.4.2 Global usage

A composition infrastructure can store if an available contributor is in use. As composition infrastructures that support run-time reconfiguration need to remove contributors during reconfiguration, they use such a registry to keep track of which contributors are in use. An example for such a system is *OSGi*. Using a tracker, a host can detect when a contributor is added and start to use it. A host can also detect when a contributor is removed and stop to use it (cf. Figure 18). OSGi keeps track of the used contributors by counting the *getService* (a host starts using a contributor) and *ungetService* (a host stops using a contributor) calls.

```
class Library {
  class BooksTracker extends ServiceTracker {
    Object addingService(ServiceReference ref) {
      Books books = (Books) context.getService(ref);
      // ... store books contributor for later use ...
      return books;
    }
    void removedService(ServiceReference ref, Object service) {
      Books books = (Books) service;
      context.ungetService(ref);
      // ... remove books contributor ...
    }
  }
}
```

Figure 18: Host that retrieves its contributors from a registry which stores global contributor usage

3.1.4.3 Host-specific availability

Composition infrastructures can store per host which contributors are available to them. The composition infrastructure can retrieve this information, e.g., from a configuration file. Figure 19 shows such a configuration file for Spring [Johnson et al., 2011] and our library application. The *Library* host is configured to get the contributors *LocalBooks* and *OffsiteBooks*. When *Library* is instantiated, Spring injects the configured contributors into the *Library*'s constructor. Other hosts can be configured with a different set of contributors (not shown).

```
<beans>
  <bean class="Library">
    <constructor-arg>
      <list>
        <bean class="LocalBooks" />
        <bean class="OffsiteBooks" />
      </list>
    </constructor-arg>
  </bean>
</beans>
```

Figure 19: Host that retrieves its contributors from a registry which stores host-specific contributor availability

3.1.4.4 Host-specific usage

Composition infrastructures that support dynamic reconfiguration can keep track of which contributor instances are used by which specific hosts. Hosts can use this information to share contributors; tools can use it to, for example, to visualize the composition of the program, to store a snapshot of the composition, or to reconfigure the program. To demonstrate this, we show a composition state visualizer based on *Plux* [Wolfinger, 2010]. Plux maintains the composition state in the instance store, i.e., it stores the extensions and their connections. The visualizer component retrieves the composition state from the *instance store* and draws a graph with the extensions and their connections (cf. Figure 20).

```
[Extension]
class Visualizer {
  Extension self;
  Visualizer(Extension self) {
    this.self = self;
  }
  void DrawCompositionGraph() {
    InstanceStore store = self.Runtime.InstanceStore;
    foreach (Extension e in store.Extensions) {
      var connectionsToHosts = e.Plugs;
      var connectionsToContributors = e.Slots;
      // ... draw extension and connections ...
    }
  }
  // ... user interface, etc. ...
}
```

Figure 20: Component that retrieves the composition from a registry which stores host-specific contributor usage

3.1.5 Contributor cardinality

Hosts can request contributors in different cardinalities. A host can request a single mandatory contributor, a single optional contributor, or multiple contributors. Let us look at an example for each scenario.

3.1.5.1 Single mandatory contributor

A host that requests a single mandatory contributor depends on this contributor and cannot be used without it. We demonstrate this scenario with a *C* [Kernighan and Ritchie, 1988] implementation of the library host. In this example, the library host depends on the local book store and therefore statically binds the local books contributor, i.e., it includes the corresponding header file and initializes the contributor (cf. Figure 21).

```
#include "LocalBooks.h"
int main(int argc, char **argv) {
  LocalBooksInit();
  // ... use the local book store ...
}
```

Figure 21: Host that depends on a single mandatory contributor

3.1.5.2 Single optional contributor

A host that requests a single optional contributor can be used with or without this contributor. We demonstrate this scenario with a *C* [Kernighan and Ritchie, 1988] implementation of the library host using a Windows dynamic link library [Petzold, 1998]. The library host uses the optional offsite book store if it is available, i.e., if the *LoadLibrary* call succeeds. Otherwise, it falls back to the local book store (cf. Figure 22).

```
#include <Windows.h>
#include "LocalBooks.h"
#include "OffsiteBooks.h"
int main(int argc, char **argv) {
  HINSTANCE offsite = LoadLibrary("OffsiteBooks.dll");
  if (offsite != NULL) {
    OffsiteBooksInit *init = (OffsiteBooksInit)
        GetProcAddress(offsite, "OffsiteBooksInit");
    (*init)();
    // ... use the offsite book store ...
  } else {
    LocalBooksInit();
    // ... use the local book store ...
  }
}
```

Figure 22: Host that can be extended with a single optional contributor

3.1.5.3 Multiple contributors

A host that requests multiple contributors can be used without any contributor, with a single contributor, or with multiple contributors. We demonstrate this scenario with a *PicoContainer* [PicoContainer, 2012] implementation of the library host. In this example, the library host uses the contributors which are injected into

its constructor. If no contributors are injected it falls back to the local book store (cf. Figure 23).

```
class Library {
  Library(Books[] bookStores) {
    if (bookStores.length > 0) {
      // ... work with the book stores ...
    } else {
      // ... fall back to local book store ...
    }
  }
}
```

Figure 23: Host that retrieves multiple contributors

3.2 Classification of composition mechanisms

The composition mechanism of a composition infrastructure determines how hosts and contributors are connected. A composition mechanism is defined by the contributor provision characteristics described in Section 3.1. In this section, we define the following classes of composition mechanisms:

Compile-time binding	..	The components are bound at compile time and deployed as a single entity.
Run-time binding	..	The components are deployed in multiple entities. The hosts bind their contributors at run time.
Startup-time lookup	..	The contributors register their provided contracts. When a host is instantiated, it retrieves the contributors for its requested contracts from the registry.
Startup-time injection	..	The hosts register their requested contracts and the contributors register their provided contracts. When a host is instantiated, the composition infrastructure injects the contributors, which fulfill the requested contracts into the host.
Run-time lookup	..	Similar to startup-time lookup, but further contributors can become available while the host is running.
Run-time lookup with notification	..	Similar to run-time lookup, but contributors can be removed at run time and the composition infrastructure notifies the hosts upon changes in contributor availability.
Run-time injection	..	Similar to startup-time injection, but further contributors can be injected at run time. Also similar to run-time lookup with notification, but the composition infrastructure can inject a different set of contributors into each host.

Run-time injection with tracking ..	Similar to run-time injection, but the composition infrastructure keeps track of which contributors are used by each host.

Figure 24 shows an overview of the composition mechanisms with their contributor provision characteristics. The following subsections explain the composition mechanisms in detail with their specific contributor provision characteristics.

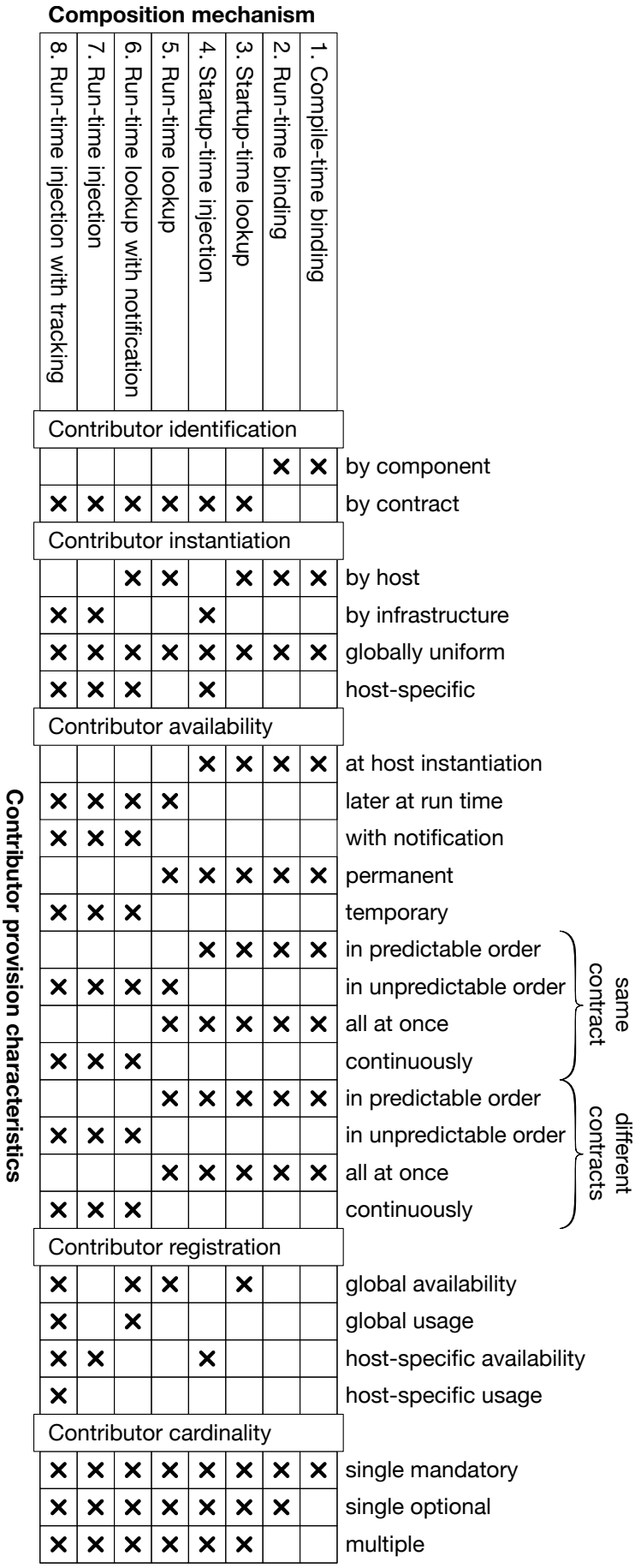


Figure 24: Composition mechanisms classified by their contributor provision characteristics

3.2.1 Compile-time binding

The compile-time binding composition mechanism composes a program at compile time. The composed program has a monolithic architecture and is deployed as a single entity. After such a program is built and deployed, it cannot be extended with further components nor can anything be removed from it. On every run, exactly the same code is executed. Typical representatives of compile-time binding systems are C/C++, Java, and .Net.

Figure 25 shows the Java implementation of the library example. The local books and offsite books contributors are identified by component name and instantiated by the library host. The *new* operator always creates a new instance and thus is a globally uniform instantiation mechanism. The contributors are referenced in the source code, linked at compile time, and thus available to the host at any time, i.e., the host can use the contributors starting from host instantiation time and it can do so permanently until the program quits. As the source code of the host determines the order in which the contributors are created, the host can rely on this order and can access all contributors at once. As one *new* statement always corresponds to one contributor, the cardinality is single and mandatory. When the program is built using the *javac* and *jar* commands, the contributors are statically linked to the library host.

```
// Build program with:
//   java Library.java LocalBooks.java OffsiteBooks.java
//   jar -cf Library.jar Library.class LocalBooks.class
//                       OffsiteBooks.class
class Library {
  LocalBooks local;
  OffsiteBooks offsite;
  Library() {
    local = new LocalBooks();
    offsite = new OffsiteBooks();
    // ... create the user interface ...
  }
}
```

Figure 25: Library application composed with compile-time binding

3.2.2 Run-time binding

The run-time binding composition mechanism composes a program in two steps. The set of contributors comprising the program is chosen at compile time; however the actual composition is done at startup time. The program can have optional contributors if it has fallback implementations for contributors which are unavailable at startup time. Similar to compile-time binding, the composed program is monolithic and cannot be extended beyond the set of contributors chosen at compile time. In contrast to compile-time binding, the program is deployed in

multiple files and a contributor file can be removed from the program. Typical representatives of run-time binding systems are Windows dynamic link libraries [Petzold, 1998] and Unix shared objects [Committee, 1995].

Figure 26 shows the C implementation of a host that uses dynamic link libraries. The local books library is a mandatory contributor, thus the host fails if it is unavailable. In contrast, the offsite books library is optional, i.e., the host uses it if it is available; if it is unavailable the host works with the local books library only.

```
#include <Windows.h>
#include "LocalBooks.h"
#include "OffsiteBooks.h"
int main(int argc, char **argv) {
  HINSTANCE local, offset;
  local = LoadLibrary("LocalBooks.dll");
  if (local == NULL) {
    return 1;
  }
  // ... use the local book store ...
  offsite = LoadLibrary("OffsiteBooks.dll");
  if (offsite != NULL) {
    // ... use the offsite book store ...
  }
}
```

Figure 26: Library application using one mandatory and one optional contributor, composed with the run-time binding composition mechanism

3.2.3 Startup-time lookup

In programs that use the startup-time lookup composition mechanism hosts identify their contributors by contract. The deployed program comprises the hosts and the contributors as well as a configuration that assigns contributors to each contract. At startup time, the hosts look up the available contributors for their contracts that were assigned to these contracts in the configuration. The configuration is global, i.e., for a requested contract every host retrieves the same set of contributors. The configuration supports different cardinalities by specifying a single contributor or multiple contributors for a contract in the configuration. If a host provides a fallback when no contributor is configured the contributor is optional, otherwise it is mandatory. A technique to compose programs with startup-time lookup is described by Fröhlich and Schwarzinger [2005, 2006].

Figure 27 shows the C# implementation of a library host which uses startup-time lookup to request a book store contributor by contract. The *Books* class is the contract for book store contributors as well as the factory for them. Fröhlich and Schwarzinger call this combination of contract and factory a connector. When the *Books* connector is loaded it looks up the contributor in the configuration file (cf. Figure 27c), i.e., it retrieves the assembly and the type of the configured contribu-

tor and instantiates it. In our example, this is the *LocalBooks* contributor (cf. Figure 27b). The library host (cf. Figure 27d) retrieves the contributor from the static method *Get* in the class *Books* (cf. Figure 27a).

a)
```
class Books {
  static Books books;
  static Books() {
    try {
      string assemblyName =
          ConfigurationManager.AppSettings["BooksProvider"];
      string booksName =
          ConfigurationManager.AppSettings["BooksClass"];
      Assembly booksAssembly = Assembly.LoadFrom(assemblyName);
      Type type = booksAssembly.GetType(booksName);
      books = (Books) Activator.CreateInstance(type);
    } catch (Exception) {
      // ... load a default implementation ...
    }
  }
  public static Books Get() {
      return books;
  }
  // ... abstract book store methods ...
}
```

b)
```
class LocalBooks : Books {
  // ... book store method implementations ...
}
```

c)
```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <appSettings>
    <add key = "BooksProvider" value = "LocalBooks.dll" />
    <add key = "BooksClass" value = "LocalBooks" />
  </appSettings>
</configuration>
```

d)
```
class Library {
  static void Main(string[] args) {
    Books books = Books.Get();
    // ... use the book store ...
  }
}
```

Figure 27: Library host retrieving the contributors as specified in a configuration file, using the startup-time lookup composition mechanism

3.2.4 Startup-time injection

In programs that use the startup-time injection composition mechanism the hosts just declare which contracts they request. The composition infrastructure instantiates the contributors and injects them into the hosts. The deployed program comprises hosts, contributors, and a configuration. The configuration determines which contributors are injected into which hosts, thus the contributors can be provided in a host-specific way. The contributors can be injected into the con-

structor of a host, into the fields of the host object, or via method calls. Representatives of startup-time injection systems are dependency injection containers such as *PicoContainer* [PicoContainer, 2012], *Spring* [Johnson et al., 2011], and *Microsoft Unity* [Microsoft, 2010a].

Figure 28 shows a Spring implementation of the library host which gets its contributors by constructor injection. The Spring infrastructure reads the desired composition from the configuration file (cf. Figure 28a), instantiates the *LocalBooks* contributor and injects it into the *Library* host via the constructor (cf. Figure 28b). Please note, that other hosts which request the same contract can be injected with a different set of contributors.

a)

```
<beans>
  <bean class="Library">
    <constructor-arg>
      <bean class="LocalBooks" />
    </constructor-arg>
  </bean>
</beans>
```

b)

```
class Library {
  Books bookStore;
  Library(Books bookStore) {
    this.bookStore = bookStore;
    // ... create the user interface ...
  }
  // ... main method, etc. ...
}
```

Figure 28: Library host that uses startup-time injection to obtain its contributors as specified in a configuration file

3.2.5 Run-time lookup

In programs that use the run-time lookup composition mechanism the hosts retrieve the contributors for a contract from the composition infrastructure. As the composition infrastructure can add contributors at run time, the retrievable set of contributors can be different between two retrievals. By this, run-time lookup supports dynamic additions. However, whether a specific host supports dynamic additions depends on its implementation. A host that retrieves contributors at several points in time supports dynamic addition, whereas a host that retrieves the contributors only once (e.g., at startup time) does not. Typical representatives for run-time lookup are the Java service loader [Oracle, 2006] and the Microsoft Component Object Model [Microsoft, 1995].

Figure 29 shows the Java service loader implementation of the library host. When the library is instantiated, it retrieves the initial set of contributors for the *Books* contract from the service loader. Later during operation, it continually asks the service loader to retrieve an updated set of contributors. Thus contributors that

have been added to the class path in the meantime will appear in the contributor set.

```
class Library {
  Set<Books> bookStores = new HashSet<Books>();
  ServiceLoader<Books> bookServices;

  Library() {
    bookServices = ServiceLoader.load(Books.class);
    updateBookStores();
    // ... create the user interface ...
  }

  void updateBookStores() {
    bookServices.reload();
    for (Books b : bookServices) {
      bookStores.add(b);
    }
  }
  // ... main method, etc. ...
}
```

Figure 29: Library host supporting dynamic additions by continually looking up the contributors using the run-time lookup composition mechanism

3.2.6 Run-time lookup with notification

In programs that use the run-time lookup with notification composition mechanism the hosts retrieve the contributors for a contract from the composition infrastructure. The composition infrastructure can add and remove contributors at run time. On such changes, it notifies the hosts. A host that reacts to these notifications by retrieving the updated set of contributors supports dynamic additions. Typical representatives for run-time lookup with notification are the *OSGi* framework [2011], *Eclipse* [2003], and *NetBeans* [Boudreau et al., 2007].

Figure 30 shows the OSGi implementation of the library host. In the constructor, the library host retrieves the initial set of contributors from the service tracker. During operation, the host uses the service tracker to listen to the change notifications of the composition infrastructure. On an addition notification, the host retrieves the added contributor and starts using it. Vice-versa, on a removal notification, the host stops using the contributor and gives it back.

```
class Library {
  List<Books> bookStores;
  BooksTracker tracker;

  class BooksTracker extends ServiceTracker {
    BooksTracker(BundleContext context) {
      super(context, Books.class.getName(), null);
    }
    Object addingService(ServiceReference ref) {
      Books books = (Books) context.getService(ref);
      bookStores.add(books);
      return books;
    }
    void removedService(ServiceReference ref, Object service) {
      Books books = (Books) service;
      bookStores.remove(books);
      context.ungetService(ref);
    }
  }

  Library(BundleContext context) {
    bookStores = new CopyOnWriteArrayList<Books>();
    tracker = new BooksTracker(context);
    tracker.open();
    for (ServiceReference ref : tracker.getServiceReferences()) {
      Books b = (Books) tracker.getService(ref);
      bookStores.add(b);
    }
  }
  // ... main method, etc. ...
}
```

Figure 30: Library host that updates its contributors at run time using the run-time lookup with notification composition mechanism

3.2.7 Run-time injection

Composition infrastructures with the run-time injection composition mechanism inject the contributors into the host in the same way as startup-time injection does. However, run-time injection does so continuously while the host is running. The host can get contributors via method calls and through constructor or field injection. In the case of constructor and field injection, the host usually uses an observable collection of contributors and monitors it for changes. Typical representatives for run-time injection are the *Microsoft Managed Extensibility Framework* (MEF, [Microsoft, 2010b]) and the *Plux* composition infrastructure [Wolfinger, 2010].

Figure 31 shows an implementation of the library host, which uses MEF to get its contributors. Hosts and contributors (both are called *parts*) declare their provided (called exports) and required (called imports) contracts using attributes. The *Books* contract is exported by the local books contributor (cf. Figure 31a) and imported by the library host (cf. Figure 31b). The library host declares the import in a

way so that it requests multiple books contributors and allows recomposition. The main method of the program (cf. Figure 31c) initiates the composition. It creates a composition container and composes the parts *Library* and *LocalBooks*. During composition, MEF fills the *Books* import of the library host with the matching *Books* export of the *LocalBooks* contributor. When new parts are added to the container later at run time, the composition infrastructure recomposes the program, i.e., it injects the new contributors into the *BookStores* property.

a)
```
interface Books {
  // ... book store methods ...
}
```

```
[Export(typeof(Books))]
class LocalBooks : Books {
  // ... book store method implementations ...
}
```

b)
```
class Library {
  [ImportMany(AllowRecomposition=true)]
  IEnumerable<Books> BookStores { get; set; }
  void UseBookStores() {
    foreach (Books bookStore in BookStores) {
      // ... use book store ...
    }
  }
}
```

c)
```
class Application {
  static void Main(string[] args) {
    var library = new Library();
    var container = new CompositionContainer();
    container.ComposeParts(library, new LocalBooks());
    // ... use the library ...
  }
}
```

Figure 31: Library host that gets its contributors from the composition infrastructure using the run-time injection composition mechanism

3.2.8 Run-time injection with tracking

Composition infrastructures with the run-time injection with tracking composition mechanism maintain the connections between the hosts and their contributors in the program, i.e., they keep track of which hosts use which contributors. This affects the host implementation: with run-time injection with tracking, the hosts do not need to store their contributors themselves, instead they can retrieve them from the composition infrastructure on demand. When the composition infrastructure changes the contributors at run time, the hosts automatically have the new contributors, because they retrieve their contributors on every access instead of maintaining a copy internally. A representative for run-time injection with tracking is the *Plux* composition infrastructure [Wolfinger, 2010].

Figure 32 shows a Plux implementation of the library host. Hosts and contributors declare their provided and requested contracts using attributes. The library host (cf. Figure 32a) uses a *slot* to specify that it requests contributors for the contract *Books* (cf. Figure 32b). The local book store (cf. Figure 32c) uses a *plug* of kind *Books* to specify that it is such a contributor. The Plux default is that slots request multiple contributors and allow recomposition. At startup Plux performs the initial composition as follows: it starts a core extension, which has an *Application* slot; it fills the *Application* slot with the library (which has an *Application* plug); it fills the *Books* slot of the library with the *Books* plug of the *LocalBooks* extension. When Plux detects that contributors are added or removed at run time, it updates the composition state accordingly. These composition changes are reflected in the hosts immediately.

a)
```
[Extension]
[Plug("Application")]
[Slot("Books")]
class Library : IApplication {
  Slot booksSlot;
  Library(Extension self) {
    booksSlot = self.Slots["Books"];
  }
  void UseBookStores() {
    foreach (Plug p in booksSlot.PluggedPlugs) {
      Books bookStore = (Books) p.Extension.Object;
      // ... use book store ...
    }
  }
}
```

b)
```
[SlotDefinition("Books")]
interface Books {
  // ... book store methods ...
}
```

c)
```
[Extension]
[Plug("Books")]
class LocalBooks : Books {
  // ... book store method implementations ...
}
```

Figure 32: Library host that gets its contributors from the composition infrastructure using the run-time injection composition mechanism

3.3 Composability fault classification

Every composition mechanism (cf. Section 3.2) has specific contributor provision characteristics (cf. Section 3.1). These characteristics may lead to a specific set composability faults, which fall into the following classes:

Host faults	..	Are faults that cause errors in hosts because the host implementation does not comply with the contributor provision characteristics of the used composition mechanism (cf. Sections 3.3.1 - 3.3.5).
Contributor faults	..	Are faults in contributors that cause errors in correctly implemented hosts, because the contributor implementation does not correspond to the composition standard of the composition infrastructure (cf. Section 3.3.6).
Composition standard violations	..	Are faults that cause errors in components, because the component implementation violates the composition standard. Such a fault can cause an error in the faulty component itself or in other components. Since examples for composition standard violations can only be given in the context of a specific composition standard, we refer to Chapters 4 and 5: Chapter 4 presents the Plux composition standard. Chapter 5 presents faults in the Plux context, including composition standard violations.

3.3.1 Contributor cardinality faults

A contributor cardinality fault is a mismatch between the cardinality supported by a host (host cardinality) and the cardinality composed by a composition mechanism (composed cardinality). Ordered from least to most flexible, the contributor cardinalities are: single mandatory, single optional, and multiple. A contributor cardinality mismatch causes an error, if the host cardinality is less flexible than the composed cardinality, otherwise the mismatch does not cause errors. For example, a host cardinality of single mandatory can cause an error with a composed cardinality of single optional, namely if the composition mechanism cannot provide a contributor. In contrast, the reverse scenario does not cause errors, because the single optional host works flawlessly with an always provided contributor.

The following subsections show examples for the possible mismatch scenarios.

3.3.1.1 Single mandatory vs. single optional

A host with single mandatory cardinality causes errors if it is composed without a contributor. This can happen with the composition mechanisms that use single optional or multiple cardinality (cf. composition mechanisms 2 to 8 in Figure 24 on page 34).

Figure 33 shows a C implementation of the library host with single mandatory cardinality for the offsite book store contributor. The host treats the contributor as mandatory, by using the contributor without checking the result of the load library call. If the composition mechanism does not provide a contributor, this host fails.

This host is composed with the run-time binding composition mechanism, which composes with single mandatory cardinality if the *OffsiteBooks.dll* is available and with single optional otherwise. Thus if the dll is unavailable the host fails, because the load library call does not provide the contributor.

```
#include <Windows.h>
#include "OffsiteBooks.h"
int main(int argc, char **argv) {
  HINSTANCE offsite = LoadLibrary("OffsiteBooks.dll");
  OffsiteBooksInit *init = (OffsiteBooksInit)
      GetProcAddress(offsite, "OffsiteBooksInit");
  (*init)();
  // ... use the book store ...
}
```

Figure 33: Library host with single mandatory cardinality that fails if composed with a single optional cardinality composition mechanism

3.3.1.2 Single mandatory vs. multiple

A host with single mandatory cardinality also causes errors if it is composed without a contributor (cf. Section 3.3.1.1), as well as if it is composed with more than one contributors. More than one contributor can be provided by composition mechanisms that use multiple cardinality (cf. composition mechanisms 3 to 8 in Figure 24 on page 34).

Figure 34 shows a Plux implementation of the library host with single mandatory cardinality for the book store contributor. The host treats the contributor as mandatory by using it in the methods *GetBook* and *Dispose* without a prior null reference check. Furthermore, as the host stores the contributor in a field, it supports only a single contributor. If the composition mechanism composes the host with more than one contributor, the host overwrites the field and uses only the last contributor composed. Moreover, it handles the book stores incorrectly, because it opens each book store when the contributor is connected, but closes only the last one, when the host itself is disposed. As the other book stores remain open, their system resources stay improperly allocated.

Plux uses run-time injection with tracking composition, which composes with multiple cardinality, i.e., it will provide all available book store contributors to the host. Thus, if more than one contributor for the books contract is available at run time, the fault in the host will result in an error, i.e., in unclosed book stores.

Plux composes the host with multiple cardinality, i.e., it connects all available book store contributors to the host by calling *SetBookStore* repeatedly. Thus, if more than one contributor is available for the books contract, the fault in the host causes erroneously unclosed book stores.

```
[Extension]
[Plug("Application")]
[Slot("Books", OnPlugged = "SetBookStore")]
class Library : IApplication, IDisposable {
  Books bookStore;
  void SetBookStore(CompositionEventArgs args) {
    bookStore = (Books) args.Plug.Extension.Object;
    bookStore.Open();
  }
  void Dispose() {
    bookStore.Close();
  }
  Book GetBook(int bookId) {
    return bookStore.GetBook(bookId);
  }
}
```

Figure 34: Library host with single mandatory cardinality that fails if a multiple cardinality composition mechanism composes with zero or with more than one contributors

3.3.1.3 Single optional vs. multiple

A host with single optional cardinality works without a contributor, but causes errors if it is composed with more than one contributor. With composition mechanisms that use multiple cardinality, both scenarios can occur (cf. composition mechanism 3 to 8 in Figure 24 on page 34).

Figure 35 shows a modified Plux implementation of the library host from Figure 34. In contrast to the host in the previous example, this host uses a statically linked fallback book store if no contributor is connected. Thus the host works also if no book store is connected; however, it still causes the same errors as described in Section 3.3.1.2 if multiple contributors are connected.

```
[Extension]
[Plug("Application")]
[Slot("Books", OnPlugged = "SetBookStore")]
class Library : IApplication, IDisposable {
  Books fallback = new ...
  Books bookStore;
  void SetBookStore(CompositionEventArgs args) {
    bookStore = (Books) args.Plug.Extension.Object;
    bookStore.Open();
  }
  void Dispose() {
    if (bookStore != null) {
      bookStore.Close();
    } else {
      fallback.Close();
    }
  }
  Book GetBook(int bookId) {
    if (bookStore != null) {
      return bookStore.GetBook(bookId);
    } else {
      return fallback.GetBook(bookId);
    }
  }
}
```

Figure 35: Library host with single optional cardinality that fails if a multiple cardinality composition mechanism composes more than one contributor

3.3.2 Contributor availability faults

A contributor availability fault is a mismatch in terms of time, order, and duration of contributor availability between how a host expects contributors and how a composition mechanism provides contributors. Time means that contributors become available at host instantiation time or later at run time. Order means that contributors become available in predictable or unpredictable order, as well as that they become available all at once or continuously. Duration means that contributors are available permanently or temporarily. These categories imply the following faults:

Time fault .. A contributor availability time fault causes an error, if a host expects the contributors to be available at host instantiation time whereas the composition mechanism provides them only later at run time.

Order fault .. A contributor availability order fault causes an error, if a host expects the contributors in a predictable order whereas the composition mechanism provides them in an unpredictable order. Order also causes an error, if a host expects all contributors to become available at once whereas the composition mechanism provides them continuously at run time.

Duration fault .. A contributor availability duration fault causes an error, if a host expects contributors to be available permanently whereas the composition mechanism provides the contributors only temporarily.

The following subsections show examples for the mismatch scenarios grouped by time, order, and duration faults.

3.3.2.1 Time faults

A host that expects the contributors to be available at host instantiation time causes errors if its contributors are not available at this time. These errors can occur with mandatory contributors as well as with optional contributors: in the mandatory case, the host fails, because it tries to access a contributor which was unavailable at instantiation time, even if it becomes available later. In the optional case, the host fails, because it uses only the contributors that are available at instantiation time and faultily neglects the contributors that become available later.

The following subsections show examples for the possible scenarios with a mismatch between the time a host expects the contributors and the time a composition mechanism provides them.

3.3.2.1.1 Availability at host instantiation time vs. later at run time

Figure 36 shows an OSGi implementation of the library host that expects a book store contributor to be available when the constructor is executed. In the constructor, the host requests the book store from the service tracker. If the book store is unavailable at this time, the tracker returns a null reference and the host fails when the method *getBook* is executed.

OSGi uses two composition mechanisms, run-time lookup and run-time lookup with notification. This host implementation is incomplete, because it looks up the contributors only at instantiation time. Neither does it update the contributors at run time, nor does it react to the notifications from OSGi. Thus the host fails, if OSGi provides the contributor later at run time.

```
class Library {
  Books bookStore;
  Library(BundleContext context) {
    ServiceTracker tracker = new ServiceTracker(context,
        Books.class.getName(), null);
    tracker.open();
    bookStore = (Books) tracker.getService();
  }
  Book getBook(int bookId) {
    return bookStore.getBook(bookId);
  }
}
```

Figure 36: Library host with time fault that fails if the composition mechanism makes the contributor available only later at run time

3.3.2.1.2 Availability at host instantiation time vs. on notification

Figure 37 shows a modified OSGi implementation of the library host from Figure 36, which retrieves multiple book stores from the service tracker. Like the previous host, this host fails if the contributors are unavailable at instantiation time. This host has a second time fault, because it neglects the contributors that become available later at run time.

```
class Library {
  Books[] bookStores;
  Library(BundleContext context) {
    ServiceTracker tracker = new ServiceTracker(context,
        Books.class.getName(), null);
    tracker.open();
    Object[] services = tracker.getServices();
    bookStores = Arrays.copyOf(services, services.length,
        Books[].class);
  }
  Book getBook(int bookId, int storeIndex) {
    return bookStore[storeIndex].getBook(bookId);
  }
}
```

Figure 37: Library host with time fault that neglects the contributors that the composition mechanism makes available later at run time

3.3.2.2 Order faults

A host with order faults causes errors if the host expects the contributors to become available in a specific order, but the composition mechanism composes them in a different order. A host also has an order fault if it expects the contributors in an arbitrary but fixed order on every run, but the composition mechanism composes them in a different order for each run. Furthermore, a host has an order fault if it expects the contributors to become available all at once, but the composition mechanism composes them continuously. A host can have order faults

when handling contributors for the same contract as well as when handling contributors for multiple contracts.

The following subsections show examples for the possible scenarios with a mismatch between the order a host expects the contributors to be composed and the order a composition mechanism provides them.

3.3.2.2.1 Predictable order vs. unpredictable order (same contract)

Figure 38 shows an implementation of the library host, which uses the Java service loader to request its book store contributors in the constructor. The host requests the local book store first and then the offsite book store. If the book stores are not provided in this order, the library mixes up the book stores and thus behaves unexpectedly.

The Java service loader uses the run-time lookup composition mechanism, which does not guarantee a specific contributor order for a contract. A correct implementation of this host would identify the local and offsite contributors by their metadata and assign them to the according fields.

```
class Library {
  Books localBooks, offsiteBooks;
  Library() {
    ServiceLoader<Books> bookServices =
        ServiceLoader.load(Books.class);
    Iterator<Books> iterator = bookServices.iterator();
    if (iterator.hasNext()) localBooks = iterator.next();
    if (iterator.hasNext()) offsiteBooks = iterator.next();
  }
}
```

Figure 38: Library host with order fault that expects the contributors in a specifc order (same contract)

3.3.2.2.2 Predictable order vs. unpredictable order (different contracts)

Figure 39 shows a Plux implementation of the library host, which uses two contracts, one for book stores and another one for a statistics tool. It expects the statistics contributor to be connected before the book store contributors.

The host starts using the statistics contributor in the *AddBookStore* method as soon as the first book store is connected, without a prior null reference check. If the composition mechanism composes a book store before the statistics tool, this host fails. This fault is similar to the contributor cardinality fault of expecting a contributor to be mandatory when it is optional (cf. example in Section 3.3.1.1 on page 43). However, in contrast to the cardinality fault, where a host only fails if no contributor is available, a host with the order fault fails already if the contracts are composed in an unexpected order. In other words, even if a statistics contributor

is available, the host will fail nonetheless if a book store contributor is composed before the statistics contributor.

```
[Extension]
[Plug("Application")]
[Slot("Books", OnPlugged = "AddBookStore")]
[Slot("Statistics", OnPlugged = "SetStatistics")]
class Library : IApplication {
  List<Books> bookStores = new List<Books>();
  Statistics statistics;
  void AddBookStore(CompositionEventArgs args) {
    Books b = (Books) args.Plug.Extension.Object;
    bookStores.Add(b);
    statistics.addBookStore(b);
  }
  void SetStatistics(CompositionEventArgs args) {
    statistics = (Statistics) args.Plug.Extension.Object;
  }
}
```

Figure 39: Library host with order fault that expects the contributors for different contracts in a specific order

3.3.2.2.3 Same order on every run vs. unpredictable order (same contract)

Figure 40 shows an implementation of the library host, which uses the Java service loader to request its book store contributors in the constructor, similar to the implementation in Figure 38. But in contrast to there, this library host stores the book stores in a list, instead of storing them in separate fields for the local and offsite book store. As the host does not expect a specific order in which its contributors are composed, it can work with any order. However, the import/export feature expects that on every run the contributors are composed in the same order. Let us look the following scenario: on one run, the export feature of the host writes all books from all book stores to a file. On the next run, the import feature of the host reads the books from that file and restores them into the book stores. This feature will only work correctly, i.e., restore the right books into the right book stores, if the contributors are composed in the same order on the import run as well as on the export run.

The Java service loader uses the run-time lookup composition mechanism, which does not guarantee the same contributor order on every run. A correct implementation of this host would export the books together with the information to which book store they belong, so that it could use this information during import in order to assign the books to the corresponding book store.

```
class Library {
  List<Books> bookStores;
  Library() {
    bookStores = new ArrayList<Books>();
    ServiceLoader<Books> bookServices =
        ServiceLoader.load(Books.class);
    for (Books b : bookServices) {
      bookStores.add(b);
    }
  }
  void exportBooks(ObjectOutputStream out) {
    int size = bookStores.size();
    out.writeInt(size);
    for (int store = 0; store < size; ++store) {
      Books b = bookStores.get(store);
      // ... export the books of the store ...
    }
  }
  void importBooks(ObjectInputStream in) {
    int size = in.readInt();
    for (int store = 0; store < size; ++store) {
      Books b = bookStores.get(store);
      // ... import the books of the store ...
    }
  }
}
```

Figure 40: Library host with order fault that expects the contributors in the same order on every run (same contract)

3.3.2.2.4 Same order on every run vs. unpredictable order (different contracts)

Figure 41 shows a Plux implementation of the library host, which uses two contracts, one for book stores and another one for a statistics tool. It expects the contributors for the different contracts to be composed in the same order on every run. This host works if the statistics contract is composed before the book contract on every run, as well as if the contracts are composed vice-versa. However, the host fails with a null reference error if the statistics contract is composed first on one run, and composed second on a subsequent run. On the first run, the statistics tool is available and the *UseStatistics* setting in the dictionary is set to *true*. On subsequent runs, the dictionary returns *true* for the *UseStatistics* setting, which bypasses the null reference check. The fault in this host is that the null reference check for the statistics tool is logically combined using an incorrect || operator instead of the right && operator. Furthermore, a correct host would only read the *UseStatistics* setting instead of setting it to *true*.

```
[Extension]
[Plug("Application")]
[Slot("Books", OnPlugged = "SetBookStore")]
[Slot("Statistics", OnPlugged = "SetStatistics",
      OnUnplugging = "RemoveStatistics")]
class Library : IApplication, IDisposable {
  Dictionary<String, bool> settings = LoadSettingsFromFile();
  Books bookStore;
  Statistics statistics;
  void LoadSettingsFromFile() { settings = ... }
  void SaveSettingsToFile() { ... }
  void SetBookStore(CompositionEventArgs args) {
    bookStore = (Books) args.Plug.Extension.Object;
    if (settings.ContainsKey("UseStatistics")
        && settings["UseStatistics"] || statistics != null) {
      settings["UseStatistics"] = true;
      statistics.UpdateBookCount(bookStore.Count);
    } else {
      settings["UseStatistics"] = false;
    }
  }
  void SetStatistics(CompositionEventArgs args) {
    statistics = (Statistics) args.Plug.Extension.Object;
  }
  void RemoveStatistics(CompositionEventArgs args) {
    statistics = null;
  }
  void Dispose() {
    SaveSettingsToFile();
  }
}
```

Figure 41: Library host with order fault that expects the contributors in the same order on every run (different contracts)

3.3.2.2.5 All at once vs. continuously (same contract)

Figure 42 shows a Java service loader implementation of the library host, which collects the number of ordered books from all book store contributors. The host works with any number of book stores, as long as they are available all at once.

On the first call of the *updateBooksOrdered* method, the host initializes the counter array where it stores individual counts of the book stores. Then it updates the counters with the number of books ordered in the contributors. This implementation has the following flaws: if the number of contributors increases before the next call of *updateBooksOrdered*, the host fails, because the array is too short; if the number of contributors decreases, the calculation is incorrect, because the array still contains values from removed contributors. By coincidence, if the set of contributors changes but the number of contributors remains the same, the host works correctly.

```
class Library {
  int[] booksOrdered;
  ServiceLoader<Books> bookServices;
  void init() {
    bookServices = ServiceLoader.load(Books.class);
    int count = 0;
    for (Books b : bookServices) {
      count++;
    }
    booksOrdered = new int[count];
  }
  void updateBooksOrdered() {
    if (booksOrdered == null) {
      init();
    }
    booksServices.reload();
    int index = 0;
    for (Books books : booksServices) {
      int nrBooksOrdered = 0;
      for (Book b : books.getBooks()) {
        if (b.isOrdered()) {
          nrBooksOrdered++;
        }
      }
      booksOrdered[index++] = nrBooksOrdered;
    }
  }
}
```

Figure 42: Library host with order fault that expects the contributors to be available all at once (same contract)

3.3.2.2.6 All at once vs. continuously (different contracts)

Figure 43 shows a Plux implementation of the library host, which uses a statistics tools to determine the value of the books for each book store. The host works with any number of book store contributors, as well as with any number of statistics contributors, if all book stores and statistics tools are composed at once. However, if the book stores and the statistics tools are composed continuously the host fails.

When Plux finishes the composition, i.e., when it has composed all available contributors, the host initializes a two-dimensional array in order to store values. For this purpose, the constructor of the host registers the callback method *Init* that is called by Plux when composition is done. On user request, the host executes the *Update* method, which retrieves the book values from the stores and stores them into the array. This works, as long as the number of contributors remains the same. If Plux adds another contributor, the host fails, because the *Update* method writes beyond the array's boundaries. If Plux removes contributors, the host fails, because the *Update* method does not shrink the array and keeps out-

dated values. A correct implementation of this host, would update the array's dimensions on composition changes.

```
[Extension]
[Plug("Application")]
[Slot("Books")]
[Slot("Statistics")]
class Library : IApplication {
  int[,] booksValue;
  Library(Extension self) {
    self.Runtime.Composer.InvokeOnCompositionDone(Init);
  }
  void Init() {
    int stores = Slot["Books"].PluggedPlugs.Count;
    int statistics = Slot["Statistics"].PluggedPlugs.Count;
    booksValue = new int[stores, statistics];
    Update();
  }
  void Update() {
    int stores = Slot["Books"].PluggedPlugs.Count;
    int statistics = Slot["Statistics"].PluggedPlugs.Count;
    for (int b = 0; b < stores; ++b) {
      Books bookStore = (Books) Slot["Books"]
          .PluggedPlugs[b].Extension.Object;
      for (int s = 0; s < statistics; ++s) {
        Statistics statistics = (Statistics) Slot["Statistics"]
            .PluggedPlugs[s].Extension.Object;
        booksValue[b, s] = statistics
            .calculateBooksValue(bookStore);
      }
    }
  }
}
```

Figure 43: Library host with order fault that expects the contributors to be available all at once (different contracts)

3.3.2.3 Duration faults

A host with duration faults causes errors if it expects the contributors to be available permanently after provision, but the composition mechanism composes them only for temporary use. Such a host fails if the composition mechanism removes a contributor. Please note, that the inverse scenario is not error-prone, because a host that expects temporary contributors also works with permanent contributors.

Figure 44 shows a Plux implementation of the contributor *LocalBooks* and the library host, which expects the book store contributor to be available permanently after its provision. The host stores its book store contributor in a field when it is composed. For this purpose, it registers the *SetBookStore* callback method for the *Plugged* event of Plux. On user request, the host executes the *GetBook* method, which uses the contributor stored in the field. When Plux removes the

book store contributor and the host accesses it thereafter, the host causes a run-time error, because Plux already disposed *LocalBooks* after unplugging it from the library. A correct implementation of the host would register a callback for the *Unplugged* event, which sets the field *bookStore* to *null* and thus causes the host to stop using the removed contributor.

```
[Extension]
[Plug("Application")]
[Slot("Books", OnPlugged = "SetBookStore")]
class Library : IApplication {
  Books bookStore;
  void SetBookStore(CompositionEventArgs args) {
    bookStore = (Books) args.Plug.Extension.Object;
  }
  Book GetBook(int bookId) {
    if (bookStore != null) {
      return bookStore.GetBook(bookId);
    }
    return null;
  }
}
[Extension]
[Plug("Books")]
class LocalBooks : Books, IDisposable { ... }
```

Figure 44: Library host with duration fault, which expects the contributors to be available permanently after provision and thus fails if a temporary contributor is removed

3.3.3 Contributor identification faults

A contributor identification fault is a mismatch between how a host identifies its contributors, and how the composition mechanism identifies them. Possible ways of identification are by component and by contract. A host with an identification fault identifies contributors by contract, but expects specific contributors. Such a host causes an error if the composition mechanism composes other than the expected contributors.

Figure 45 shows an implementation of the library host, which uses the Java service loader to request its book store contributor in the constructor. It uses the *Books* interface to identify the contributor by contract. When it retrieves the contributor instance, it expects the specific *LocalBooks* component by casting the instance to the component's class. However, if the Java service loader provides a different component for the *Books* contract, the host fails with a type cast exception. A correct implementation of this host would use the contributor as a general *Books* component and not as a specific *LocalBooks* component.

```
class Library {
  LocalBooks bookStore;
  Library() {
    ServiceLoader<Books> bookServices =
        ServiceLoader.load(Books.class);
    Iterator<Books> iterator = bookServices.iterator();
    localBooks = (LocalBooks) iterator.next();
  }
}
```

Figure 45: Library host with identification fault, which expects a specific contributor and thus fails if a different contributor is provided

3.3.4 Contributor instantiation faults

A contributor instantiation fault is a mismatch in terms of who creates a contributor (i.e., the host itself or the composition mechanism), or how contributors are created (i.e., uniformly for every host or in a host-specific way). The following subsections show examples for both mismatch scenarios.

3.3.4.1 By host vs. by infrastructure

Depending on the used composition mechanism, either the host or the composition infrastructure is responsible for creating contributor instances. This makes the following fault scenarios possible: a host wrongly creates the contributors itself, whereas it should use the contributors provided by the composition infrastructure (overeager host); or the host should create the contributors itself, but omits to do so (lazy host). This section shows an example for an overeager host. It does not show a lazy host example, because lazy hosts are not found in practice for two reasons: since lazy hosts never work, regardless of the composition scenario, developers unavoidably detect the fault; furthermore, committing the fault is hard in the first place, because if a composition infrastructure does not provide contributors, developers obviously realize that the host must create them itself.

Figure 46 shows a Plux implementation of the contributor *LocalBooks* and an overeager library host, which instantiates its contributors itself when they are composed. For this purpose, it registers the *AddBookStore* callback method for the *Plugged* event of Plux. In this method it retrieves the type name of the contributor and uses reflection to instantiate the contributor. By doing so, the host creates a new instance instead of using the instance provided by Plux. As the *LocalBooks* has no default constructor, the host fails when it calls *CreateInstance.* A correct implementation of the host would retrieve the instance created by Plux (instead of the type name) from the arguments.

```
[Extension]
[Plug("Application")]
[Slot("Books", OnPlugged = "AddBookStore")]
class Library : IApplication {
  List<Books> bookStores = new List<Books>();
  void AddBookStore(CompositionEventArgs args) {
    Type type = Type.GetType(
        args.Plug.Extension.ExtensionType.TypeName);
    Books b = (Books) Activator.CreateInstance(type);
    bookStores.Add(b);
  }
}
[Extension]
[Plug("Books")]
class LocalBooks : Books {
  LocalBooks(Extension self) { ... }
  ...
}
```

Figure 46: Library host with instantiation fault, which fails because it creates a contributor itself instead of using the contributor connected to it

3.3.4.2 Globally uniform vs. host-specific

Globally uniform instantiation means that all contributors are instantiated in the same way (e.g., the same shared instance is provided for every host, or a separate instance is instantiated for each host). In contrast to that, host-specific instantiation means that the contributors can be instantiated differently for each host (e.g., some hosts share contributors, others do not). Since host-specific instantiation is more flexible than globally uniform instantiation, faults can only occur, if a host assumes globally uniform instantiation, but is composed with host-specific instantiation.

In composition mechanisms with host-specific instantiation, one way of instantiation is usually the default (shared or non-shared). If a host expects that sharing or non-sharing is the globally uniform instantiation mechanism, whereas it is only the default for a host specific instantiation, this host will work as long as it is only composed with hosts that use the default, but will fail if it is composed with hosts that use the non-default way of instantiation.

Figure 47 shows a Plux implementation of an offsite book store, a local book store, and a statistics tool. Each book store uses a statistics tool to accumulate the total price of its books. The offsite book store relies on the default instantiation mechanism of Plux, which composes a non-shared statistics contributor. Assuming that the offsite book store is the only host that uses the statistics contributor, everything works fine. The *OffsiteBooks.GetTotalPrice* method adds the individual book prices to the statistics tool and retrieves the total price by calling the parameterless *StatisticsTool.GetTotalPrice* method. The offsite book store can coexist with other book stores and will work flawlessly, as long as these other

book stores also use a separate instance of the statistics contributor. However, the offsite book store fails if it coexists with local book store, because *LocalBooks* shares the statistics tool of *OffsiteBooks*. Plux supports host-specific instantiation using *composition behaviors* (see Section 4.7). The local book store uses such a composition behavior to intercept the composition and to retrieve the same instance of the statistics tool as the offsite book store. The local book store also adds the prices of its books to the statistics tool, which modifies the total price retrieved by the offsite book store. As the local book store is aware that the statistics tool is shared, it uses the *StatisticsTool.GetTotalPrice* method with the *storeId* parameter (passing the unique id that Plux issues for all extensions) in order to retrieve only its own prices. The fault in the offsite book store is the use of the parameterless *StatisticsTool.GetTotalPrice* method. Please note, that the fault only causes an error when the offsite book store coexists with the local book store.

```
[Extension]
[Plug("Books")]
[Slot("Statistics")]
class OffsiteBooks : Books {
  Extension self;
  OffsiteBooks(Extension e) {
    self = e;
  }
  int GetTotalPrice() {
    var stat = (Statistics) self.Slot("Statistics").PluggedPlugs[0]
        .Extension.Object;
    foreach (Book book in ...) {
      stat.Add(self.Id, book.price);
    }
    return stat.GetTotalPrice();
  }
}

[Extension]
[Plug("Books")]
[Slot("Statistics")]
class LocalBooks : Books {
  LocalBooks(Extension e) {
    ...
    e.Slots["Statistics"].Behaviors.Add(new ...);
  }
  ...
  int GetTotalPrice() {
    ...
    return stat.GetTotalPrice(self.Id);
  }
}

[Extension]
[Plug("Statistics")]
class StatisticsTool : Statistics {
  void Add(int bookStoreId, int price) { ... }
  int GetTotalPrice(int bookStoreId) { ... }
  int GetTotalPrice() { ... }
}
```

Figure 47: Library host with instantiation fault, which expects that contributors are instantiated in a globally uniform way, and thus fails if contributors are instantiated in a host-specific way

3.3.5 Contributor registration faults

A contributor registration fault is a mismatch in terms of where a composition mechanism makes contributors available (i.e., globally to all hosts or specifically to individual hosts) or how a composition mechanism tracks contributor usage (i.e., by storing just a global usage counter per contributor or by keeping track of which hosts are connected to which contributors). The following subsections show examples for both mismatch scenarios.

3.3.5.1 Global availability vs. host-specific availability

Different composition mechanisms offer different flexibility on how they make contributors available. Some can make contributors only available globally to all hosts (less flexible), whereas others can make them available only to specific hosts (more flexible). A host that expects the more flexible availability also works with the less flexible availability. However, a host that expects less flexibility than what the composition mechanism provides can fail.

Figure 48 shows a Plux implementation of the library host and a statistics tool contributor. The library host uses the statistics tool and the available book store contributors to calculate the average price of a book. The statistics tool uses the available book stores to sum up the total price of all books. The library host divides this total price by the number of books, which are stored in the book stores composed with it. Assuming that the book store contributors are made globally available, i.e., the same set of book stores is connected to the library host and to the statistics tool, the correct average price is calculated. However, if the composition mechanism composes the library host and the statistics tool with different book store sets, the calculated average is incorrect. A correct implementation of the host would use the *GetTotalPrice* method with the *storeId* parameter to calculate the average for each book store individually and exclude book stores unavailable to the statistics tool.

```
[Extension]
[Plug("Application")]
[Slot("Books")]
[Slot("Statistics")]
class Library : IApplication {
  int CalculateAverageValue() {
    Slot s = self.Slot("Books");
    int count = 0;
    foreach (Plug p in s.PluggedPlugs) {
      var books = (Books) p.Extension.Object;
      count += books.Count;
    }
    var stat = (Statistics) self.Slots["Statistics"]
        .PluggedPlugs[0].Extension.Object;
    return stat.GetTotalPrice() / count;
  }
}
[Extension]
[Plug("Statistics")]
[Slot("Books")]
class StatisticsTool : Statistics {
  int GetTotalPrice(int bookStoreId) {
    int total = 0;
    foreach (Plug p in self.Slots["Books"].PluggedPlugs) {
      var b = (Books) p.Extension.Object;
      if (bookStoreId == -1 || p.Extension.Id == bookStoreId) {
        foreach (Book b in books.GetBooks()) {
          total += b.Price;
        }
      }
    }
    return total;
  }
  int GetTotalPrice() { return GetTotalPrice(-1); }
}
```

Figure 48: Library host with registration fault, which expects that contributors are made available globally, and thus fails if contributors are made available only to specific hosts

3.3.5.2 Global usage vs. host-specific usage

Host-specific usage causes the same hosts to fail as host-specific availability does. Usage and availability are insofar similar, as the result (the host is not using the contributor) is the same. Regardless of if the host could but does not use the contributor (usage) or the host cannot use the contributor, because it is not available (availability). Usage and availability differ only in the way the causing composition is reached: host-specific availability hides contributors from the host, whereas host-specific usage just omits to connect or disconnects contributors. Therefore, the example from Figure 48 also applies here.

3.3.6 Contributor sharing faults

A sharing fault in a contributor causes errors in a correctly implemented host, if a contributor that does not support sharing is shared among hosts by the composition mechanism. Figure 49 shows a Plux implementation of a book store contributor, which expects that the composition mechanism creates dedicated instances for each host. If the composition mechanism shares an instance of this contributor among hosts, this contributor fails and thus also its hosts fail. When the book store contributor is composed, it opens a database connection and stores it in a field. When the contributor is removed, it closes the database connection and sets the field to *null*. Assuming that the contributor is only connected to a single host the contributor works correctly. However, if the composition mechanism shares the contributor between two hosts, the *OpenDatabase* callback method is called twice: on the second call, the contributor overwrites the value in the *connection* field with the second connection. Opening a second database connection can already cause an error, depending on whether the database management system allows multiple connections or not. Similarly, when the composition mechanism disconnects the contributor from the sharing hosts, the *CloseDatabase* method is also called twice: on the first call, the (secondly opened) connection is closed and the field is nulled; on the second call, the contributor causes a null pointer error. Furthermore, after the composition mechanism removed the contributor from one host, the other host can no longer use the book store, because calling the *GetBooks* method also raises a null pointer error. A correct implementation of the contributor would check the value of the connection field and open the connection only once; and it would only close the connection when it is removed from the last host.

```
[Extension]
[Plug("Books", OnPlugged="OpenDatabase", OnUnplugging="CloseData-
base")]
class LocalBooks : Books {
  SqlConnection connection;
  void OpenDatabase(CompositionEventArgs args) {
    connection = new SqlConnection(...);
    connection.Open();
  }
  void CloseDatabase(CompositionEventArgs args) {
    connection.Close();
    connection = null;
  }
  Book[] GetBooks(String author) {
    SqlCommand command = new SqlCommand("SELECT * ...", connection);
    SqlDataReader reader = command.ExecuteReader();
    ...
  }
}
```

Figure 49: Book store contributor with sharing fault, which expects that dedicated contributor instances are created for each host, and thus fails if instances are shared among hosts

Chapter 4: Plux composition infrastructure

We implemented our testing and debugging methods based on the Plux composition infrastructure. As Plux uses the most flexible composition mechanism, namely run-time injection with tracking, Plux programs are well suited to demonstrate the possible composability faults and the application of the testing and debugging methods.

Plux is a composition infrastructure for extensible and customizable plugin-based programs [Wolfinger, 2010]. It supports *plug-and-play composition*, allowing programs to be automatically assembled from components without programming or configuration. Plux also supports *dynamic composition*, allowing programs to be dynamically reconfigured by adding, removing, or swapping sets of components at run time. An implementation of Plux is currently available for .Net [Plux, 2012], but its concepts can be ported to other platforms such as Java as well.

Plux differs from other plugin systems [Birsan, 2005] such as OSGi [OSGi, 2011] or Eclipse [Eclipse, 2003]: it provides a composer, which maintains a global composition state; it uses an event-based programming model; and it provides an exchangeable component discovery mechanism. The composer replaces programmatic composition with automatic composition. Programmatic composition, as for example in Eclipse, means that the host has to query a contributor registry and to create and integrate its contributors itself. Automatic composition, as in Plux, means that the contributors just declare their requests and provisions using metadata; the composer uses these metadata to match requests and provisions and connects matching components. Plux maintains all components an their connections in a global composition state, i.e., it keeps track of which hosts use which contributors. Hosts retrieve their contributors from the composition state. Optionally, components can react to events sent by the composer, e.g., if they want to reflect composition changes in the user interface immediately. Component discovery is the process of detecting components and extracting their metadata. The discovery mechanism is not an integral part of Plux, but a plugin itself, which makes it replaceable. The following subsections cover the characteristics of Plux in detail.

4.1 Metadata

Plux uses the metaphor of extensions, slots and plugs (cf. Figure 50). 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 another extension, it declares a *slot*. Such an extension is called a *host*. If an extension wants to provide a service to other extensions, it declares a *plug*. Such an extension is called a *contributor*. Several related extensions can be packaged as a *plugin*, which is a dll file that can be deployed and loaded separately.

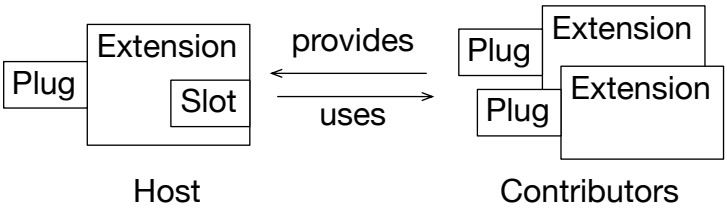


Figure 50: Metadata for Plux extensions with slots and plugs

Slots and plugs are identified by names. A plug matches a slot, if they have the same name. If so, the plug can be connected to the slot. A slot represents an interface, which has to be implemented by a matching plug (more specifically, by its corresponding class). 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 their respective slot definitions (cf. Figure 51).

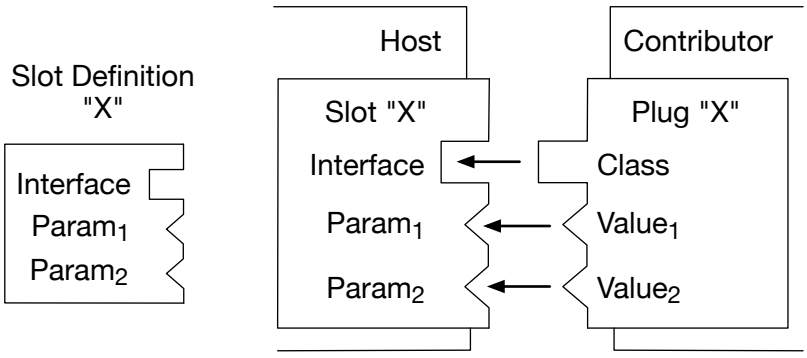


Figure 51: Metadata of a slot and plug named "X"

The means to provide metadata is customizable in Plux. The default mechanism extracts metadata from .Net attributes in dll files. Attributes are pieces of information, which can be attached to .Net constructs, such as classes, interfaces, methods, or fields. At run time, the attributes can be retrieves using reflection [ECMA, 2006].

Plux has the following custom attributes: The *SlotDefinition* attribute to tag an interface as a slot definition, the *Extension* attribute to tag a class that implements a component, the *Slot* attribute to declare requests in hosts, the *Plug* attribute to

declare provisions in contributors, the *ParamDefinition* attribute to declare required parameters in a slot definition, and the *Param* attribute to specify provided parameter values in contributors.

Let us now look at the source code of the library example from Section 3.2.8 as it is implemented in Plux. Assume that the library works with books, which it retrieves from book stores. The book store is implemented as a contributor, which plugs into the library. The slot for the book store is defined in Figure 52. A book comprises, e.g., a title, an author, and a price. The book store provides, for example, a collection of books and a method to calculate the total price of all books in the book store. Furthermore, the slot has a parameter *Kind*, to specify if it holds local books or offsite books.

```
[SlotDefinition("Books")]
[ParamDefinition("Kind", typeof(StoreKind))]
interface Books {
  Book[] Books { get; }
  int GetTotalPrice() { ... }
  ...
}
interface Book {
  string Title { get; }
  ...
}
enum StoreKind { Local, Offsite }
```

Figure 52: Interface and metadata for Plux slot definition

Figure 53 shows the implementation of the local book store contributor, which is a book provider. Since *LocalBooks* has a *Books* plug, it has to implement the interface specified in the slot definition *Books* and to provide a value for the parameter *Kind*.

```
[Extension]
[Plug("Books")]
[Param("Kind", StoreKind.Local)]
class LocalBooks : Books { ... }
```

Figure 53: Implementation and metadata for the contributor extension

Figure 54 shows the implementation of the library host. In order to be able to use a data source, the host has a *Books* slot. It also has an *Application* plug that fits into the *Application* slot of the Plux core. At startup, Plux creates an instance of *Library* and connects it to the core. It also creates an instance of *LocalBooks* and connects it to *Library*.

```
[Extension]
[Plug("Application")]
[Slot("Books")]
class Library : IApplication { ... }
```

Figure 54: Implementation and metadata for the host extension

4.2 Composition

Composition is the process that matches the requests of hosts with the provisions of contributors. In Plux, this is done by the composer. The composer assembles programs from extensions available in a plugin repository (a folder in the file system), by connecting the plugs of the contributors with the slots of the hosts.

When the user adds a new extension to the plugin repository, the composer integrates it into the program on-the-fly. Similarly, if an extension is removed from the repository, the composer removes it from the program.

Integrating an extension as contributor means that the composer instantiates it and connects its plugs with the matching slots of the extensions in the program. If a plug is connected to a slot, this relationship is called *plugged*. If the extension is also a host, i.e., if it has slots, the composer will plug matching plugs into these slots.

Removing an extension means that the composer unplugs all instances of this extension from the slots where they are plugged, i.e., it removes the *plugged* relationship for the corresponding slots and plugs.

The composer distinguishes between shared and unique contributor instances. A *unique* contributor is connected to just a single slot, whereas a *shared* contributor can be plugged into several slots. For every extension, Plux holds exactly one instance as the dedicated shared instance. Slots can declare whether they want the composer to plug them this shared instance or a new unique instance.

4.3 Composition state

Since the composer establishes all connections between components, Plux knows the instantiated extensions, their slots and plugs, as well as their connections. These data comprise the *composition state*. If a host wants to use its plugged contributors, it can query 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 (cf. Figure 55). For every slot, the composition state keeps track of which plugs are connected to it, and for every plug the composition state keeps track of to which slots it is connected.

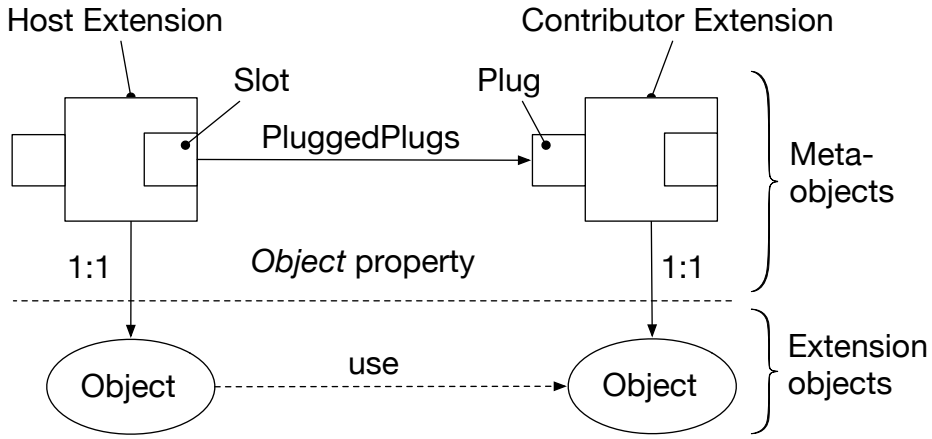


Figure 55: Meta-objects for extensions and their connections in the Plux composition state

Figure 56 describes the host from Figure 54 on page 66 in more detail, showing how meta-objects can be used by a program. When the composer creates the *Library* extension, it passes the *Library* meta-object to the constructor, which uses it to retrieve the meta-object of the *Books* slot. When the user refreshes the library, the method *RefreshListView* retrieves the contributors using the *PluggedPlugs* property of the *Books* slot. For each plugged book store, it retrieves the extension object of this store and populates the list view with the books provided.

```
[Extension]
[Plug("Application")]
[Slot("Books")]
class Library : IApplication {
  Slot booksSlot;
  ListView view = ...
  Library(Extension self) {
    booksSlot = self.Slots["Books"];
  }
  void RefreshListView() {
    foreach (Plug p in booksSlot.PluggedPlugs) {
      Books store = (Books) p.Extension.Object;
      foreach (Book b in store.Books) {
        view.Add(b.Title);
      }
    }
  }
  ...
}
```

Figure 56: Retrieving meta-objects and contributors from the Plux composition state

To complete the example, we compile the slot definition interface *Books* to a dll file, the so-called *contract*, and the classes *Library* and *LocalBooks* to plugin dll files. From these files, Plux composes the program as shown in Figure 57.

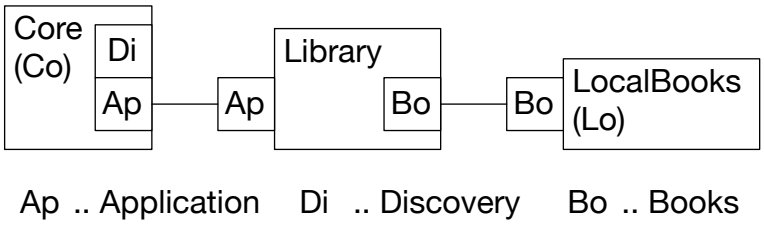


Figure 57: Plux composition state of the library example

4.4 Composition events

In addition to querying the composition state, a host can listen to composition events from the composer. This is suitable for a host that must immediately react to added or removed contributors, e.g., to update its user interface. Figure 58 shows a modified version of our host from Figure 56. The modified version uses the *Slot* attribute to register event handler methods for the *Plugged* and the *Unplugged* event. In this example, the event handlers just print out which book store (its extension name and the value of the *Kind* parameter) was plugged and unplugged. Their *args* parameter holds information about the composition event, e.g., the plug of the connecting extension.

```
[Extension]
[Plug("Application")]
[Slot("Books", OnPlugged="AddBookStore",
      OnUnplugging="RemoveBookStore")]
class Library : IApplication {
  ...
  void AddBookStore(CompositionEventArgs args) {
    Extension e = args.Plug.Extension;
    StoreKind kind = args.Plug.Params["Kind"].Value;
    Console.WriteLine("Plugged:" + e.Name + " " + kind);
  }
  void RemoveBookStore(CompositionEventArgs args) {
    Extension e = args.Plug.Extension;
    StoreKind kind = args.Plug.Params["Kind"].Value;
    Console.WriteLine("Unplugged:" + e.Name + " " + kind);
  }
  ...
  Library(Extension self) { ... }
  void RefreshListView() { ... }
}
```

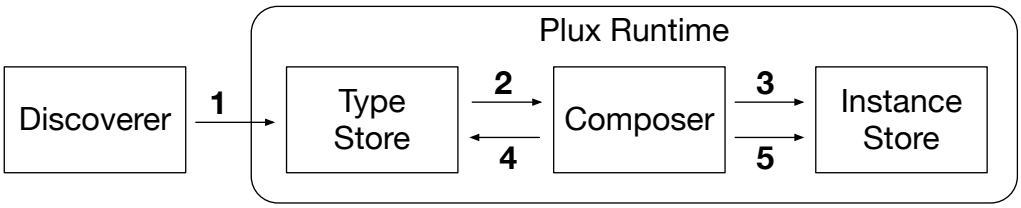
Figure 58: Reacting to composition events from the Plux composer

4.5 Composition infrastructure

The composition infrastructure builds programs from contracts and plugins. It discovers extensions from a plugin repository and composes the program from them by connecting matching slots and plugs. The plugin repository is typically a direc-

tory in the file system containing contract dll files (with slot definitions) and plugin dll files (with extensions).

Figure 59 shows the subsystems of the composition infrastructure and how they interact. The *discoverer* ensures that at any time the type store contains the metadata of all extensions and slot definitions from the plugin 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.



1 Adds and removes contracts and plugins
2 Notifies on changes
3 Queries for matching slots
4 Queries for matching plugs
5 Stores instance metadata and relationships

Figure 59: Architecture of the Plux composition infrastructure

The *type store* maintains the type 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 requests and provisions as declared in the metadata. It listens to changes in the type store and updates the program accordingly, i.e., it 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 relationships between them. The instance store is also used by tools such as the Plux visualizer, which visualizes the composition state and its changes during run time.

4.6 Programmatic composition

The process described in the previous sections, where the composer automatically connects extensions and extensions can query their connections, is called *automatic composition*. In addition to that, extensions can make connections using *programmatic composition*, i.e., the extensions can control how the composer assembles the program. For example, the extensions can use the composer's API

to connect only specific contributors, a script interpreter can assemble a program according to a script, or a deserializer can restore a previously serialized program.

Figure 60 shows a host that uses programmatic composition to connect only specific contributors depending on some user input. To do so, the host disables automatic composition by setting the *AutoPlug* parameter in the *Slot* attribute to *false*. Furthermore, to let the user control which book stores should be connected, the host provides a user interface widget to switch between local and offsite book stores. When the user switches, the *SwitchTo* method is called with a parameter that specifies the chosen book store kind (*Local* or *Offsite*). The method uses the composer to unplug all currently plugged contributors; it retrieves the available contributors for the books slot from the type store; for each contributor, it retrieves the *Kind* parameter value and checks whether it matches the chosen kind; if so, it retrieves the shared instance of this contributor from the composer and plugs it into the books slot. Please note, that this kind of composition cannot be achieved with automatic composition, because the composer would plug in all available contributors, regardless of their *Kind* parameter value.

```
[Extension]
[Plug("Application")]
[Slot("Books", AutoPlug = false)]
class Library : IApplication {
  Extension self;
  TypeStore typeStore;
  Composer composer;
  Library(Extension self) {
    this.self = self;
    composer = self.Runtime.Composer;
    typeStore = self.Runtime.TypeStore;
  }
  void SwitchTo(StoreKind kind) {
    Slot s = self.Slots["Books"];
    foreach (Plug p in s.PluggedPlugs) {
      Composer.Unplug(s, p);
    }
    foreach (PlugType pt in typeStore.GetPlugTypes("Books")) {
      if (pt.Params["Kind"] == kind) {
        Plug p = Composer.GetShared(
            pt.ExtensionType).Plugs["Books"];
        Composer.Plug(s, p);
      }
    }
  }
  ...
}
```

Figure 60: Using the composer for programmatic composition

4.7 Behavior-guided composition

In addition to programmatic composition, Plux allows customizing the composition process by using *composition behaviors*. A composition behavior is a reusable piece of code which can be applied to a slot or to the composer in order to guide composition. For example, a behavior applied to a slot can control how many contributors can be connected to this slot or it can ensure that this slot is only filled if some other slot has been filled before. A behavior applied to the composer can, for example, be used to enforce application-wide security restrictions, e.g., it can restrict the integration of extensions to those from trusted manufacturers. Plux provides predefined behaviors for common composition patterns, but developers can also define application-specific behaviors. The benefit of using declaratively specified behaviors instead of programmatic composition is that behaviors can be reused in many situations, which leads to less programming overhead.

The library host in Figure 61 uses a behavior to guide the composition in such a way that only a single contributor at a time can be connected to the books slot. In the constructor, the host creates a *SingleContributorBehavior* and applies it to the books slot. The behavior is a class that overrides the *CanPlug* method and allows the plug operation only if no contributors are plugged at that time, and denies it otherwise. To do so, it retrieves the number of plugged contributors from the slot to which it is applied. In this example, the behavior creates an informational log message, which appears in the Plux event log for diagnostic purposes. The single contributor behavior can be reused in other hosts and actually there is such a behavior in the Plux library. A behavior can suppress composition operations (as shown here), but it can also react to composition events and perform composition operations. For more details on composition behaviors, see [Jahn et al., 2010a].

```
[Extension]
[Plug("Application")]
[Slot("Books")]
class Library : IApplication {
  Library(Extension self) {
    self.Slots["Books"].Behaviors.Add(new SingleContributorBehav-
ior());
  }
}

class SingleContributorBehavior : Behavior {
  override bool CanPlug(CompositionEventArgs args, out LogArgs log) {
    if (BehaviorSlot.PluggedPlugs > 0) {
      log = new LogArgs("Contributor rejected. Host supports "
          + "single contributor only.");
      return false;
    } else {
      log = null;
      return true;
    }
  }
}
```

Figure 61: Using a behavior to guide the composition

4.8 Composition standard

The composition standard specifies the rules according to which a composition must be done in order to be valid. According to the Plux composition standard, an extension can assume the following: a contributor is only used by hosts to which it is connected (plugged); a contributor's methods are only called from a single thread (the Plux runtime thread); if an extension invokes a composition operation by calling the composer, the composition change is complete when the call returns; if such a composition operation is cancelled, an exception is thrown.

The contributor use of hosts must match the composition state, i.e., a host must intend to use every contributor which is connected to it. If a host does not intend to use some contributors, it must guide the composer so that these contributors are not connected, typically using a behavior.

From these assumptions, the following constraints must be deduced: A host must use only contributors that are plugged into it and it must use all of them. Calls between extensions must be made in the Plux runtime thread. A composition call to the composer during a composition event, must not contradict the operation that caused that event, e.g., a contributor must not be unplugged while the *Plugged* event for this contributor is handled.

Plux extensions must comply with the constraints defined by the composition standard. The composer prevents contradictory operations, by terminating them with an exception. Therefore such faults can easily be detected during develop-

ment. However, the other constraints are unchecked by Plux and thus faults may show up only after an extension has been deployed to the user, i.e., when it is connected to other extensions. In order to detect as many of these faults as possible already during development, a systematic test procedure is necessary.

Chapter 5: Finding composition errors

The contributor provision characteristics of a composition mechanism and the composition standard of a composition infrastructure determine the set of possible composability faults. A composability test method should find the resulting errors. In this chapter, we present a new testing method for the run-time injection with tracking composition mechanism, show the application of this test method to for Plux components, present a composability test tool for Plux, and present the results of an experimental evaluation of the test method and the test tool.

5.1 The automated composability test method Act

The automated composability test method (Act) tests the composability of components, which are composed with the run-time injection with tracking composition mechanism. Act is a dynamic black-box test method. It is dynamic, because it composes and executes the component under test. It is a black-box method, because it tests components without analyzing their inner workings (i.e., the source code).

5.1.1 Composability test procedure

Act connects and disconnects the component under test (testee) with different hosts and contributors and thereby varies the order in which the components are connected and disconnected. In doing so, Act tries to detect if the testee works in some composition orders but fails in others. Components can fail if they make wrong assumptions about the composition mechanism or the composition state. Act reveals wrong assumptions by making the hosts integrate and remove contributors in various orders. Depending on the implementation of the testee, errors may only show up when the testee is actually executed, i.e., when its methods are called. To reveal such errors, Act performs functional tests on the testee between composition changes.

The Act testing method (cf. flowchart in Figure 62) comprises the following steps: Step 1 generates test cases (i.e., sequences of composition operations) for the given testee and its test bed (i.e., its hosts and its contributors). Step 2 executes the test cases, i.e., for each test case it sets up the test bed, instantiates the testee, and performs the functional tests. In step 3 the composition operations of the

test case are executed. After each composition operation, the functional tests are performed on the testee. As the result of the method, the composition errors and the functional errors are returned.

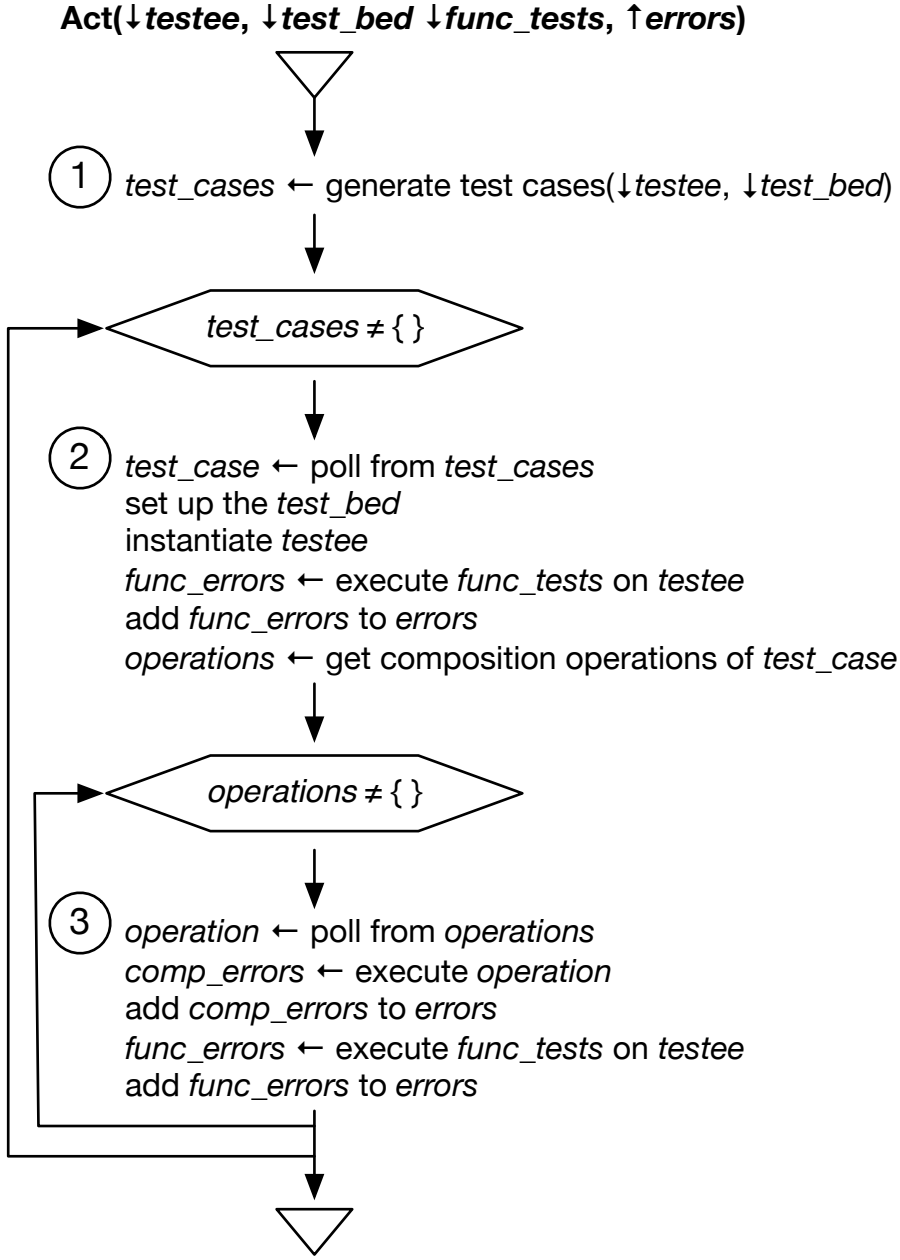
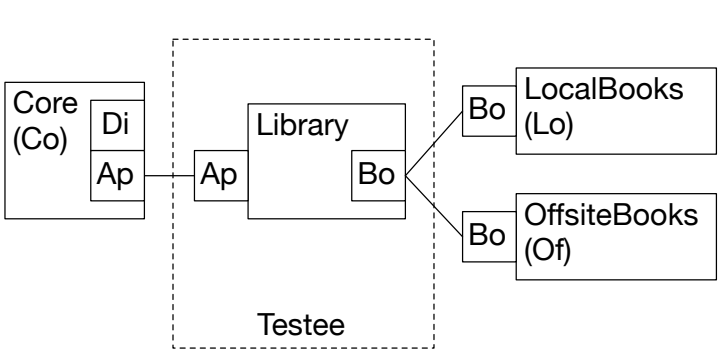


Figure 62: Flowchart of the automated composability test method Act

5.1.2 Generating test cases

Act generates test cases for the given testee and the given test bed. The test bed comprises the possible hosts and contributors of the testee. It can include actual components as well as mock components. Act generates the test cases by using *lexicographic permutation* [Knuth, 2005], i.e., it permutes the plug and unplug composition operations for the components in the test bed in every possible order, but removes invalid test cases (e.g., test cases that contain an unplug operation before the corresponding plug operation).

Figure 63 shows the test case generation for the library example. The library component is the testee. The test bed comprises the Plux *Core* as a host as well as the local book store and the offsite book store as contributors (cf. Figure 63a). Figure 63b shows the generated test cases (i.e., the generated composition operation sequences), e.g., test case 1 composes the testee and the test bed in the following order: plug testee into *Core*, plug *LocalBooks* into testee, plug *OffsiteBooks* into testee, unplug *OffsiteBooks* from testee, unplug *LocalBooks* from testee, and unplug testee from *Core*. The test cases 2 to 90 permute the order of these plug and unplug operations. The total number of test cases for this test bed is 90 (cf. Figure 63c). The total number of test cases is the total number of permutations divided by the ratio of total to valid composition orders, i.e., by 2^c where c is the number of components. The value 2 comes from the fact that there are two composition operations (plug and unplug), which might be in the wrong order (i.e., a contributor is unplugged before it was plugged). The exponent c comes from the fact that these composition operations can be in the wrong order for any of the c components. In other words, the factor is the number of permutations where at least one component would be unplugged before it is plugged. By this means the invalid composition orders are excluded. In the given example the factor is 8, because for each valid composition order there are 7 composition orders which contain an unplug-plug pair for at least one component.



Ap .. Application Di .. Discovery Bo .. Books

a) Test bed setup

Composition operations

1	Co	Lo	Of	~~Of~~	~~Lo~~	~~Co~~
2	Co	Lo	Of	~~Of~~	~~Co~~	~~Lo~~
...	...	...	...	...	...	...
89	Co	~~Co~~	Lo	~~Lo~~	Of	~~Of~~
90	Lo	~~Lo~~	Co	Of	~~Co~~	~~Of~~

Xx .. Plug ~~Xx~~ .. Unplug

b) Test cases

$$\text{numTestCases} = \frac{(2 * \text{numComponents})!}{2^{\text{numComponents}}} = \frac{(2 * 3)!}{2^3} = \frac{720}{8} = 90$$

c) Number of test cases

Figure 63: Test case generation for the library example

5.1.3 Reducing the number of test cases

The number of generated test cases grows with to the number of components in the test bed (c) with (2*c)! / 2^c. This causes long execution times for large test

beds. To reduce the execution time, Act reduces the number of test cases by selecting only those which are likely to find errors.

To reduce the number of test cases Act applies a heuristic similar to *covering arrays* ([Hartman, 2005]). While covering arrays works on parameters, Act works on subsequences of composition operations in test cases. Act splits the composition operations in the test cases in n-tuples of operations, where n is configurable. The choice of n is a trade-off between effectiveness and run-time cost, a higher n leads to more test cases and thus more execution time. For example: a test case for two components (c1, c2) comprises the plug operations p1 and p2, as well as the corresponding unplug operations u1 and u2. Using n=2, this test case is split into three 2-tuples:

Test case:	p1, p2, u1, u2
2-Tuples:	[p1 p2], [p2 u1], [u1 u2]

The heuristic selects test cases such that across all test cases each each n-tuple is covered at least once and that as few test cases as possible are selected. The heuristic is based on the assumption that if an n-tuple does not find an error in one test case, it will also find no errors in other test cases.

The rationale behind this heuristic is that an error that occurs after a certain composition operation strongly coheres with the composition state at this time and the preceding composition operations. We assume that the closer a previous composition operation is, the more influence it has on the error, i.e., the previous one has more influence than the one before the previous.

If longer tuples are used, more previous composition operations are considered, however if our assumption is true, the influence of each additional composition operation decreases. Longer tuples result in more test cases that must be executed, e.g, for three components and thus 90 generated test cases, the heuristic leaves 17 test cases using 2-tuples, 49 using 3-tuples, 85 using 4-tuples, and all 90 using 5-tuples or longer. In our experimental evaluation (cf. Section 5.4) all seeded faults could already be found with 2-tuples. Please note, if 1-tuples were used only a single test case would be selected, no matter how many components are in the test bed, and if the tuple length is equal to the test case length, all test cases are selected.

Finding the minimal set of test cases is a problem that can be solved with the Quine-McCluskey method ([Quine, 1955], [McCluskey Jr., 1956]), which finds the prime implicants of a boolean function. Our n-tuples correspond to Quine-McCluskey's *minterms*, and our test cases correspond to their *implicants*. The minimal set of implicants can be found either using trial and error or the more systematic branch-and-bound method [Petrick, 1956].

	test cases				n-tuples	1	2	~~1~~	2	~~2~~	2	~~1~~	1	1	~~2~~
						2	~~1~~	~~2~~	~~2~~	~~1~~	1	2	~~2~~	~~1~~	1
⇨	1	2	~~1~~	~~2~~		x	x	x							
	1	2	~~2~~	~~1~~		x			x	x					
	2	1	~~1~~	~~2~~				x			x			x	
⇨	2	1	~~2~~	~~1~~						x	x		x		
⇨	1	~~1~~	2	~~2~~					x			x		x	
⇨	2	~~2~~	1	~~1~~					x					x	x

a) Select test cases which contain unique n-tuples

	test cases				n-tuples	1	2	~~1~~	2	~~2~~	2	~~1~~	1	1	~~2~~
						2	~~1~~	~~2~~	~~2~~	~~1~~	1	2	~~2~~	~~1~~	1
⇨	1	2	~~1~~	~~2~~		x	x	x							
	1	2	~~2~~	~~1~~		x			x	x					
	2	1	~~1~~	~~2~~				x			x			x	
⇨	2	1	~~2~~	~~1~~						x	x		x		
⇨	1	~~1~~	2	~~2~~					x			x		x	
⇨	2	~~2~~	1	~~1~~					x					x	x

b) Select all n-tuples from selected test cases

1, 2 .. Components Xx .. Plug .. Unique n-tuples

⇨ .. Selected test cases ~~Xx~~ .. Unplug .. Covered n-tuples

Figure 64: Finding the minimal set of test cases using the Quine-McCluskey method

Figure 64 shows the application of the Quine-McCluskey method for the example above (with the components c1 and c2). In step a) we select all test cases which contain a unique n-tuple, i.e., one that is only contained in a single test case. In step b) we select all n-tuples covered by the selected test cases from step a). Finally, we must select further test cases until all n-tuples are covered. In this example all n-tuples are covered after step b) and thus none of the remaining test cases are selected.

Finding the minimal set of test cases using the Quine-McCluskey method is time consuming (exponential run-time complexity) and memory intensive for large test beds, because the matrix has as many rows as there are test cases and as many

columns as there are n-tuples. For example, for 5 components with 2-tuples there are (2*5)! / 2^5 = 113400 test cases and [(2*5)! / (2*5 - 2)!] - 5 = 85 tuples. The formula for the number of tuples is [(2*c)! / (2*c - n)!] - i, where *c* is the number of components, *n* is the tuple length, and *i* is the number of tuples that cannot be executed because the unplug operations occurs before the corresponding plug operation. To reduce time and memory consumption for the test case selection, we apply a heuristic during test case generation. Figure 65 shows that we extract the n-tuples with the composition operations, from each generated test case. A test case that contains at least one yet uncovered n-tuple is selected, a test case that contains only n-tuples that have already been covered before is suppressed. After all permutations are processed, all n-tuples are covered.

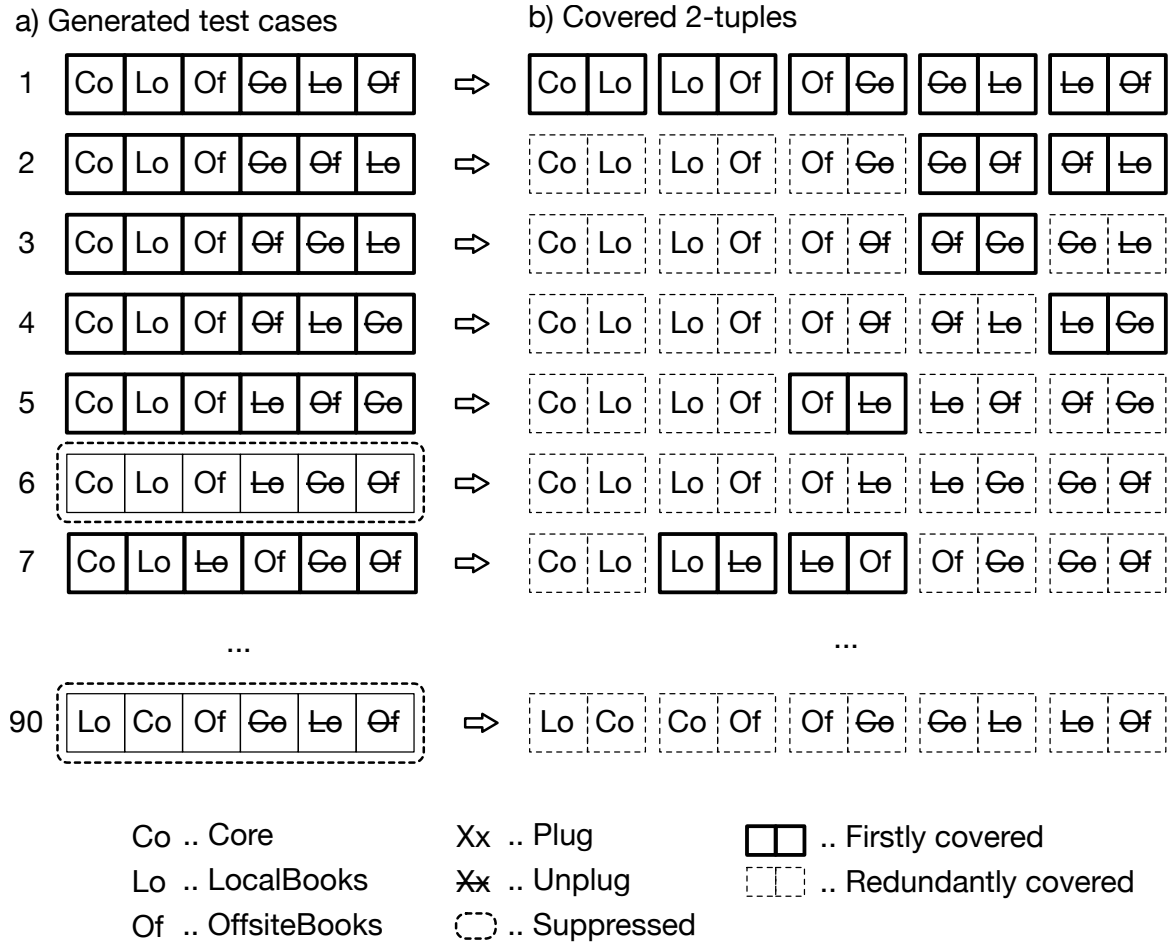
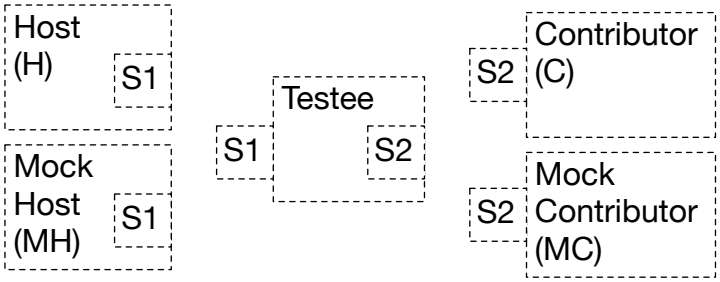


Figure 65: Generating test cases until all 2-tuples are covered

5.1.4 Specifying test beds

In order to execute the test cases for a testee the test bed of this testee must be specified. The test bed comprises the testee itself, optional host and contributor components, optional host and contributor mocks, and optional functional tests (cf. Figure 66).

a) Components



b) Mock specification

```
[Extension]                [Extension]
[Slot("S1")]               [Plug("S2")]
class MockHost { ... }     class MockContributor : S2 { ... }
```

c) Functional test specification

```
    [Test]
(1) void Test1(Testee testee) {
      Assert.AssertEquals(...);
    }
```

Figure 66: Test bed specification

Host and contributor components can be real extensions or mock extensions. In the test bed specification, the following optional attributes can be set for every component:

Use count	..	How often should an instance of this component be plugged within the execution of a test case. The default is 1.
Targets	..	The target extensions (hosts or contributors) to which this component should be connected. The default is the testee.
Count	..	The number of instances of this component that should be created during the execution of a test case. The default is 1.
Plugs	..	The names of the plugs with which this component should be plugged. This applies only to contributors with multiple plugs. The default is any plug.
Slots	..	The names of the slots of this component into which contributors should be plugged. This applies only to hosts with multiple slots. The default is any slot.
Role	..	*Host* if component is a host, *Contributor* if it is a contributor, *Both* if it is both a host and a contributor. The default is *Both*.

Functional tests comprise a method to execute and the following optional attributes which specify the required composition state when the test is executed, a maximum execution time, an expected exception, and whether the composition state should be rebuilt after the test, in detail:

Used plugs	..	The plugs of the testee that must be plugged when this functional test is executed. The default is none.
Used slots	..	The slots of the testee that must have contributors plugged when this functional test is executed. The default is none.
Plugged hosts	..	The hosts where the testee must be plugged into when this functional test is executed. The default is none.
Plugged contributors	..	The contributors which must be plugged into the testee when this functional test is executed. The default is none.
Subset	..	*True* if it suffices when a subset of the specified plugs or slots are connected to execute the functional test. Affects the attributes *Used plugs*, *Used slots*, *Plugged hosts*, and *Plugged contributors*, if multiple values are specified. The default is *False*.
Expected exception	..	The name of the exception that is expected by this functional test. The default is none.
Timeout	..	The maximum time this functional test is allowed to run. The default is infinite.
Recompose	..	*Off* executes the next functional test without recomposition, *Always* rebuilds the composition state before and after this functional test, or *OnError* rebuilds the composition state when a functional test finds an error. The default is *Off*.

5.1.5 Executing test cases

Act executes the reduced set of generated test cases for the given testee and the testbed. Figures 67 and 68 show this procedure for test case 1 of the library example. Figure 67 shows the set up phase. Step 1 polls the first test case. Step 2 polls the composition operations for test case 1. Step 3 sets up the testbed with the *Library* (testee), the Plux *Core* (host), the *LocalBooks* and the *OffsiteBooks* (contributors), and the according functional tests. A functional test is attributed with information in which composition state it must be executed. In this example, functional tests 1-3 must be executed in every composition state, functional test 4 if the *Core* is connected, functional tests 5-6 if the *LocalBooks* is connected, and functional tests 7-10 if the *OffsiteBooks* is connected.

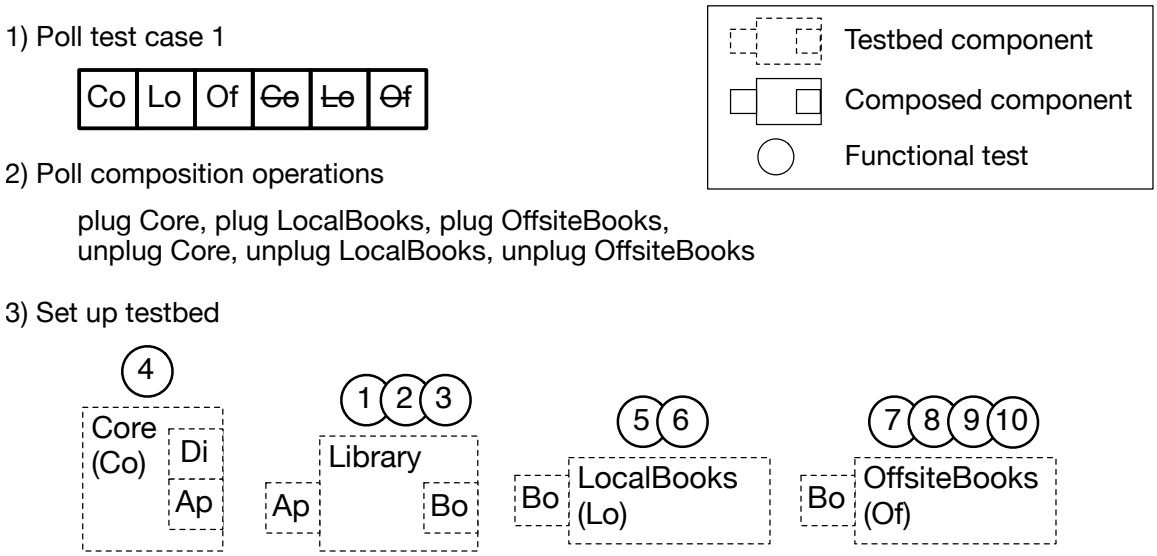
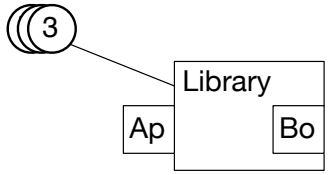


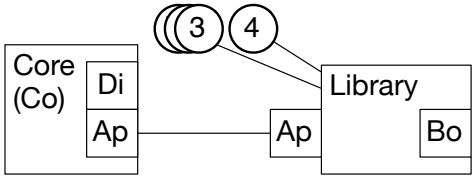
Figure 67: Setting up the testbed with components and functional tests

Figure 68 shows the execution phase. Step 4 instantiates the testee and executes functional tests 1-3. Step 5 executes the composition operation *plug Core* and executes functional tests 1-4. Step 6 plugs *LocalBooks* and executes functional tests 1-6. Step 7 plugs *OffsiteBooks* and executes functional tests 1-10. Step 8 unplugs the *Library* from the *Core* and executes functional tests 1-3 and 5-10. Step 9 unplugs *LocalBooks* and executes functional tests 1-3 and 7-10. Finally, step 10 unplugs *OffsiteBooks* and executes functional tests 1-3. Act adds the errors that occurred during composition and during the execution of functional tests to the result and repeats this procedure for all test cases in the reduced set. Please note, which composability faults are found by Act depends on the used test bed. Section 5.2 shows how to define test beds for the errors classes defined in Section 3.3.

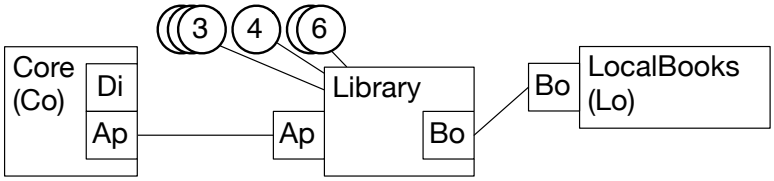
4) Instantiate testee and execute functional tests 1-3



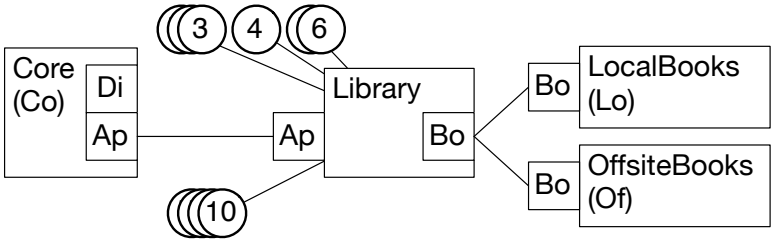
5) Execute composition operation (*plug Core*) and execute functional tests 1-4



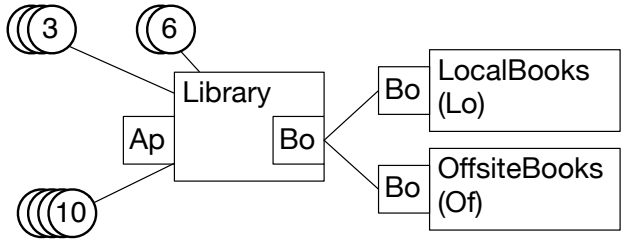
6) Plug *LocalBooks* and execute functional tests 1-6



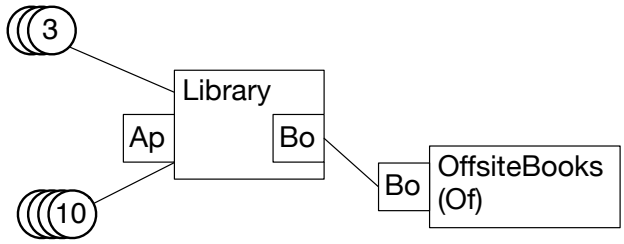
7) Plug *OffsiteBooks* and execute functional tests 1-10



8) Unplug *Core* and execute functional tests 1-3 and 5-10



9) Unplug *LocalBooks* and execute functional tests 1-3 and 7-10



10) Unplug *OffsiteBooks* and execute functional tests 1-3

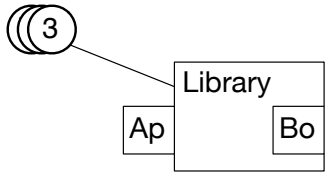


Figure 68: Executing composition operations and functional tests

5.1.6 Detecting composition standard violations

Act detects composition standard violations by wiring proxies between hosts and contributors. Figure 69 shows the *Library* host composed with a *LocalBooks* contributor: (a) shows how the extensions would be connected without Act during normal program execution (without a proxy); (b) shows how they are connected with Act during testing (with a proxy). Please note, that for shortness the proxies are omitted in the figures throughout the thesis. Act generates these proxies by analyzing slot definitions and it wires them between contributor and host when they are connected. The proxies detect if hosts use contributors that are not plugged into them, because either they have never been plugged, or they were unplugged earlier. They also detect when a host calls a contributor from an invalid thread. For more on composition standard violations see Section 5.2.7.

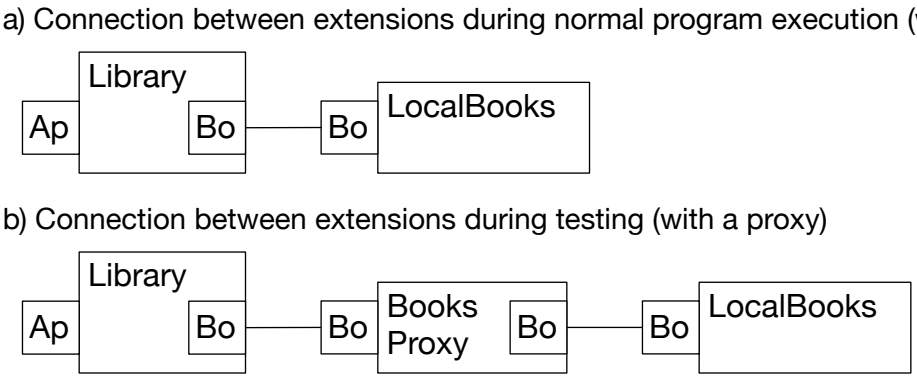


Figure 69: Composing extensions with proxies to detect composition standard violations

5.2 Finding errors in Plux components

This section shows how to apply the automated composability test method Act (cf. Section 5.1) in order to find errors in Plux components (cf. Section 3.3). Each of the following subsections gives an example of a faulty component, a testbed setup, and a test case which finds the error.

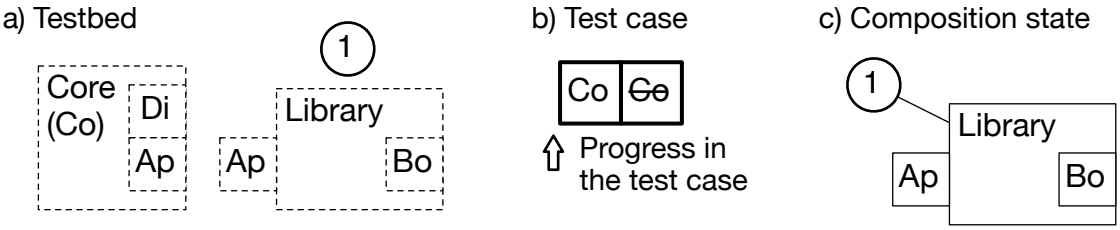
5.2.1 Contributor cardinality faults

A contributor cardinality fault is a mismatch between the cardinality supported by a host and the cardinality composed by a composition mechanism. A contributor cardinality fault can be found by varying the number of contributors, i.e., by connecting more or less contributors than the testee expects and by running functional tests against the testee. Many cardinality faults can even be found without functional tests, because they occur in composition event handlers that expect a specific number of contributors (e.g., in array boundaries) and fail al-

ready during composition (i.e., when the contributor is connected to or disconnected) if another number is composed (e.g., because the array boundaries are exceeded).

5.2.1.1 Single mandatory vs. multiple

The test case in Figure 70 composes the library host (cf. Figure 71) without any contributors. The original goal of the functional test is checking whether the library raises an *InvalidBookIdException* if a book id less than zero is passed. Then the functional test (cf. Figure 70d) is executed, the library host causes a null pointer exception. Thus the functional test (incidentally) exposes the contributor cardinality fault. The cause for this fault is that the host expects a single mandatory book store contributor, whereas the automatic composition in Plux can compose the library without a contributor, unless a composition behavior makes sure that the library is only plugged after a book store contributor was plugged into it. A correct implementation of the host would close the book store in an *OnUnplugging* handler and install a behavior on the *Books* slot, which ensures the desired cardinality of exactly one contributor at a time.



d) Functional test

```
[Test(ExpectedException=typeof(InvalidBookIdException))]
void Test(Library testee) {
  testee.GetBook(-1);
}
```

1

Figure 70: Test case that finds a single mandatory vs. multiple cardinality fault

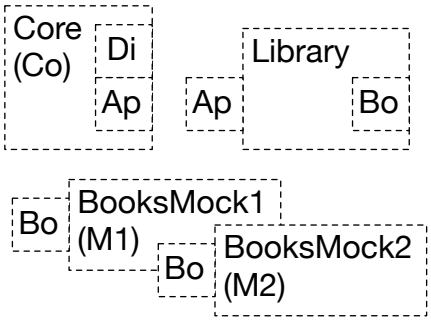
```
[Extension]
[Plug("Application")]
[Slot("Books", OnPlugged = "SetBookStore")]
class Library : IApplication {
  Books bookStore;
  void SetBookStore(CompositionEventArgs args) {
    bookStore = (Books) args.Plug.Extension.Object;
  }
  Book GetBook(int bookId) {
    return bookStore.GetBook(bookId);
  }
}
```

Figure 71: Library host with single mandatory vs. multiple cardinality fault

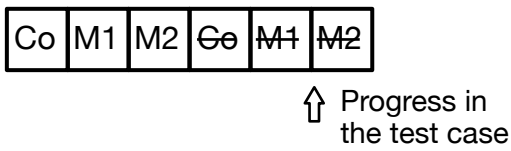
5.2.1.2 Single optional vs. multiple

The test case in Figure 72 tests the library host with two copies of a book store mock. The book store mock checks if it is closed when it is unplugged (cf. Figure 72d). When the test case progresses to the composition operation unplug *BooksMock1*, it exposes the contributor cardinality fault in the library host. The cause for the fault is that the host overwrites the reference to *BooksMock1* (stored in *bookStore*) when *BooksMock2* is plugged (cf. Figure 73) and omits to call the *Close* method on *BooksMock1*. This breaks the assertion in *BooksMock1* when the unplug event calls the *CheckIsClosed* method. A correct implementation of the host would install a behavior on the *Books* slot, which replaces the old book store when a new book store is plugged.

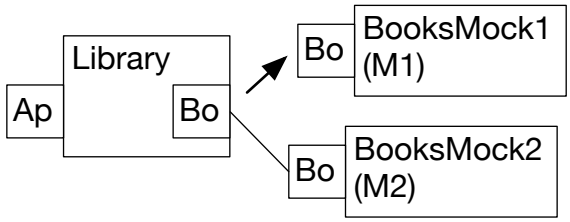
a) Testbed



b) Test case



c) Composition state



d) Mock contributor

```
[Extension]
[Plug("Books", OnUnplugged = "CheckIsClosed")]
class BooksMock : Books {
  bool isOpen;
  void Open() { isOpen = true; }
  void Close() { isOpen = false; }
  Book GetBook(int bookId) { return null; }
  void CheckIsClosed(CompositionEventArgs args) {
    Assert.AssertFalse(isOpen);
  }
}
```

Figure 72: Test case that finds a single optional vs. multiple cardinality fault

```
[Extension]
[Plug("Application")]
[Slot("Books", OnPlugged = "OpenBookStore",
      OnUnplugging = "CloseBookStore")]
class Library : IApplication {
  Books default = new ...
  Books bookStore = null;
  void OpenBookStore(CompositionEventArgs args) {
    bookStore = (Books) args.Plug.Extension.Object;
    bookStore.Open();
  }
  void CloseBookStore(CompositionEventArgs args) {
    bookStore.Close();
    bookStore = null;
  }
  Book GetBook(int bookId) {
    if (bookStore != null) {
      return bookStore.GetBook(bookId);
    } else {
      return default.GetBook(bookId);
    }
  }
}

[SlotDefinition("Books")]
interface Books {
  void Open();
  void Close();
  Book GetBook(int bookId);
}
```

Figure 73: Library host with single optional vs. multiple cardinality fault

5.2.2 Contributor availability faults

A contributor availability fault is a mismatch in terms of time, order, and duration of contributor availability between how a host expects contributors and how a composition mechanism provides contributors. A contributor availability fault can be found by varying the composition state or by varying the order in which the contributors are connected and disconnected. Some faults can only be detected by functional tests, which reveal that components have faulty behavior in certain composition states.

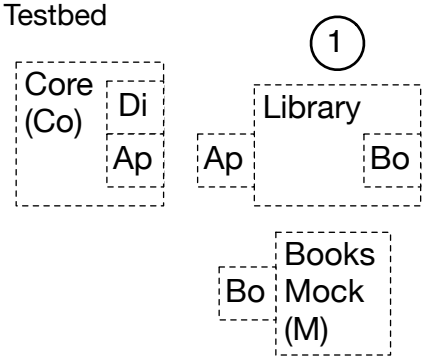
5.2.2.1 Time faults

A time fault can be found by varying the time when the testee's contributors are made available. Errors typically occur if the contributors are added later than the testee expects.

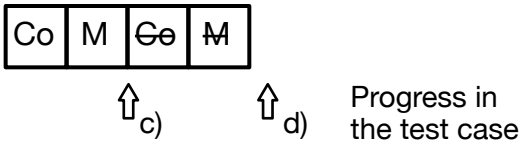
5.2.2.1.1 Availability at host instantiation vs. later at run time

The test case in Figure 74 tests the library host (Figure 75) with a book store mock. The book store mock checks if it has been used when it is unplugged (cf. Figure 74e). When the test case progresses to the composition operation plug *BooksMock*, it executes the functional test, which makes the library use the contributor. Later, when the test case progresses to the composition operation *unplug BooksMock*, it exposes the time fault in the library host. The cause for the fault is that the host does not use the books mock contributor; it uses only the contributors that it retrieved in the constructor and stored in the *bookStores* field. A correctly implemented host would retrieve the contributors from the slot on demand. The books mock exposes this misbehavior, because the assertion in the *CheckIsUsed* method breaks when the mock is unplugged.

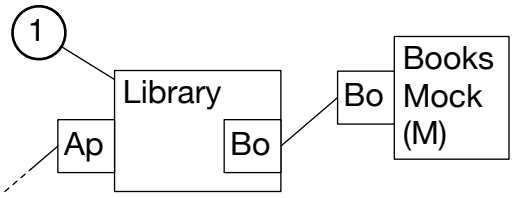
a) Testbed



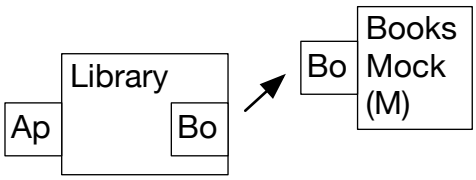
b) Test case



c) Composition state after Co, M



d) Composition state after Co, M, ~~Co~~, ~~M~~



e) Mock contributor

```
[Extension]
[Plug("Books", OnUnplugged = "CheckIsUsed")]
class BooksMock : Books {
  bool isUsed;
  Book GetBook(int bookId) { isUsed = true; return null; }
  void CheckIsUsed(CompositionEventArgs args) {
    Assert.AssertTrue(isUsed);
  }
}
```

f) Functional test

1

```
[Test]
void Test(Library testee) {
  testee.GetBook(1);
}
```

Figure 74: Test case that finds a host instantiation vs. later at run time availability fault

```
[Extension]
[Plug("Application")]
[Slot("Books")]
class Library : IApplication {
  List<Books> bookStores = new List<Books>();
  Library(Extension self) {
    var runtime = self.Runtime;
    var composer = runtime.Composer;
    var typeStore = runtime.TypeStore;
    foreach (PlugType pt in typeStore.GetPlugTypes("Books")) {
      Plug p = composer.Create(pt.ExtensionType).Plugs["Books"];
      self.Slots["Books"].Plug(p);
      bookStores.Add((Books) p.Extension.Object);
    }
  }
  Book GetBook(int bookId) {
    foreach (Books bookStore in bookStores) {
      Book b = bookStore.GetBook(bookId);
      if (b != null) {
        return b;
      }
    }
    return null;
  }
  // ...
}
```

Figure 75: Library host with host instantiation vs. later at run time availability fault

5.2.2.1.2 Availability at host instantiation time vs. on notification

In Plux, there are two ways for a host to correctly consider contributors that become available later at run time: The host can either retrieve the contributors from the slot on demand (as suggested in Section 5.2.2.1.1.), or it can react to plug notifications sent by the composer. As both ways are correct for later-at-run-time contributor integration, the example from Section 5.2.2.1.1. applies here as well.

5.2.2.2 Order faults

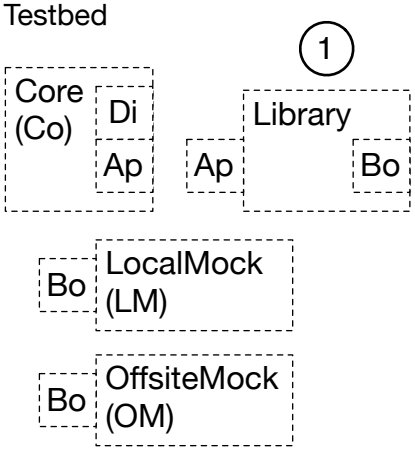
An order fault can be found by varying the order in which a testee's contributors are composed, i.e., the testee can show faulty or correct behavior with the same composition state, depending on the order of the composition operations which produced this composition state. Some order faults can be found during composition, whereas others can only be found with a functional test.

5.2.2.2.1 Predictable order vs. unpredictable order (same contract)

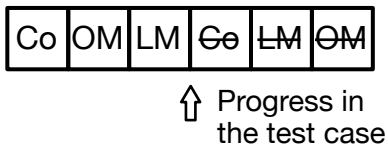
The test case in Figure 76 tests the library host (cf. Figure 77) with a book store mock for local books and a book store mock for offsite books. Combined with a functional test, the mocks check if the *GetBook* calls for local and offsite books are forwarded to the appropriate books contributor. A call with *StoreKind.Local*

should reach the *LocalMock*, and a call with *StoreKind.Offsite* should reach the *OffsiteMock*. When the test case plugs the offsite mock first and the local mock second, and executes the functional test, it exposes the order fault in the library host. The cause for the fault is that the host assumes the first plugged book store to be the local book store and the other book stores to be offsite book stores. However, the test case composed them the other way round, i.e., the offsite book store first and the local book store second. A correctly implemented host would not keep the book stores in private fields *localBookStore* and *offsiteBookStores*, but would retrieve them from the *Books* slot and use them according to the parameter values. The mocks expose the misbehavior of the host, because the assertion in the *GetBook* method breaks, when the calls from the functional test are forwarded to the wrong contributors.

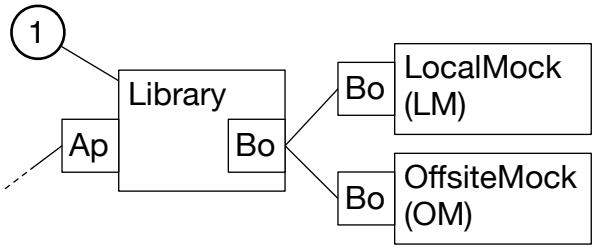
a) Testbed



b) Test case



c) Composition state



d) Local mock contributor

```
[Extension]
[Plug("Books")]
[Param("Kind", StoreKind.Local)]
class LocalMock : Books {
  Book GetBook(int bookId) {
    Assert.AssertEquals(1, bookId);
    return null;
  }
}
```

e) Offsite mock contributor

```
[Extension]
[Plug("Books")]
[Param("Kind", StoreKind.Offsite)]
class OffsiteMock : Books {
  Book GetBook(int bookId) {
    Assert.AssertEquals(2, bookId);
    return null;
  }
}
```

f) Functional test

```
[Test(PluggedContributors = { "LocalMock", "OffsiteMock"} )]
void Test(Library testee) {
  testee.GetBook(StoreKind.Local, 1);
  testee.GetBook(StoreKind.Offsite, 2);
}
```

(1)

Figure 76: Test case that finds a predictable order vs. unpredictable order fault (same contract)

```
[Extension]
[Plug("Application")]
[Slot("Books", OnPlugged = "AddBookStore")]
class Library : IApplication {
  Books localBookStore;
  List<Books> offsiteBookStores = new List<Books>();
  void AddBookStore(CompositionEventArgs args) {
    var books = (Books) args.Plug.Extension.Object;
    if (localBookStore == null) {
      localBookStore = books;
    } else {
      offsiteBookStores.Add(books);
    }
  }
  Book GetBook(StoreKind kind, int bookId) {
    Book b = null;
    switch(kind) {
      case StoreKind.Local:
        b = localBookStore.GetBook(bookId);
        break;
      case StoreKind.Offsite:
        foreach (Books books in offsiteBookStores) {
          Book b = books.GetBook(bookId);
          if (b != null) {
            break;
          }
        }
        break;
    }
    return b;
  }
}

[SlotDefinition("Books")]
[ParamDefinition("Kind", typeof(StoreKind))]
interface Books { ... }

enum StoreKind { Local, Offsite }
```

Figure 77: Library host with predictable vs. unpredictable order fault (same contract)

5.2.2.2.2 Predictable order vs. unpredictable order (different contracts)

The test case in Figure 78 tests the library host (cf. Figure 79) with the local books store and the statistics tool. When the test case progresses to the *plug Local-Books* operation, it exposes the order fault in the library host. The cause for this fault is a null pointer exception when the host accesses the uninitialized field *statistics*. A correct implementation of this host would, for example, apply a behavior on the books slot which makes sure that it is only filled after the *Statistics* slot has been filled. Such a behavior would return *false* in *CanPlug* until the statistics slot is filled.

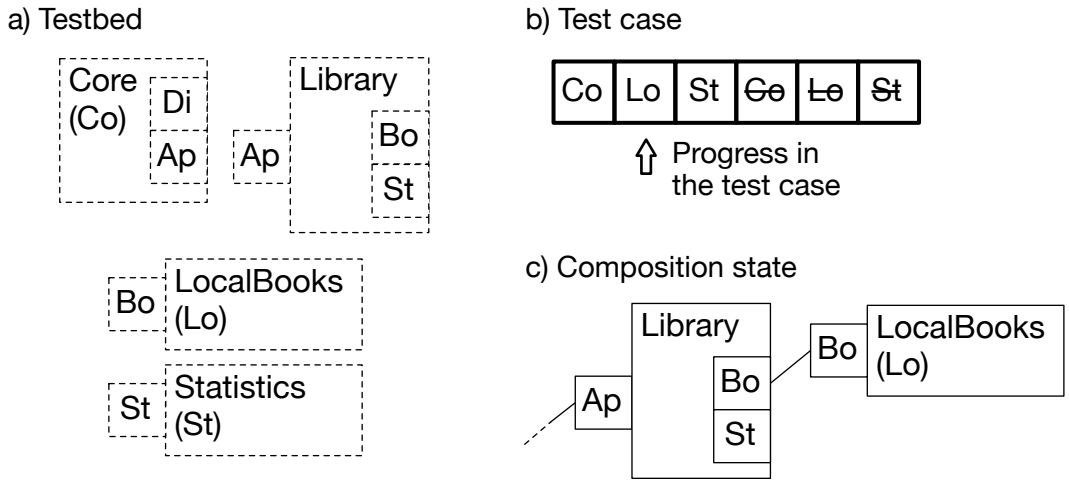


Figure 78: Test case that finds a predictable order vs. unpredictable order fault (different contracts)

```
[Extension]
[Plug("Application")]
[Slot("Books", OnPlugged = "AddBookStore")]
[Slot("Statistics", OnPlugged = "SetStatistics")]
class Library : IApplication {
  List<Books> bookStores = new List<Books>();
  Statistics statistics;
  void AddBookStore(CompositionEventArgs args) {
    Books b = (Books) args.Plug.Extension.Object;
    bookStores.Add(b);
    statistics.AddBookStore(b);
  }
  void SetStatistics(CompositionEventArgs args) {
    statistics = (Statistics) args.Plug.Extension.Object;
  }
}
```

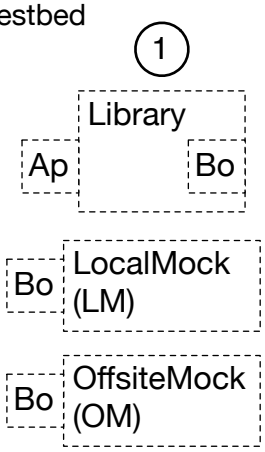
Figure 79: Library host with predictable vs. unpredictable order fault (different contracts)

5.2.2.2.3 Same order on every run vs. unpredictable order (same contract)

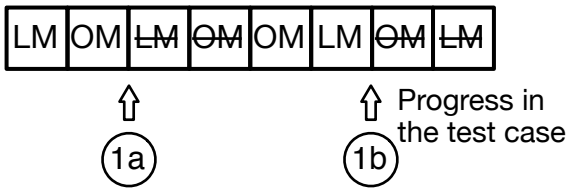
The test case in Figure 80 test the library host (cf. Figure 81) with a book store mock for local books and a book store for offsite books. Combined with a functional test, the mocks check if the library imports the books into the same book stores where they were exported from. The functional test is executed as soon as both book mocks have been plugged. The test case plugs the book mocks twice: on the first time, the functional test saves the books to a memory stream; on the second time, it restores the books from the memory stream. When the mocks are plugged in again (this time in different order), the assertions in the book mocks fail, because the restored books are different from the saved books. Thus the functional test exposes the order fault in the library host. The cause for the order fault is that the library exports and imports the data in the order in which the book stores were plugged. If they were plugged in different orders at export and import

time, the restored data are incorrectly assigned. A correct implementation of the host would identify the contributors by their component name and would assign the data accordingly.

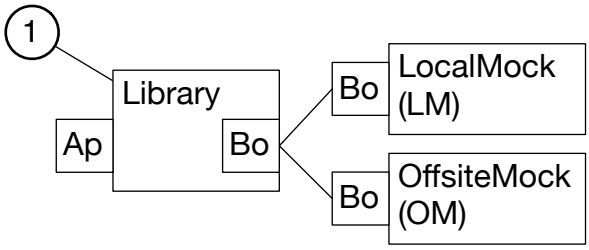
a) Testbed



b) Test case



c) Composition state



d) Local mock contributor

```
[Extension]
[Plug("Books")]
class LocalMock : Books {
  Book[] GetBooks() {
    return new Book[] {
        new MockBook(1) };
  }
  void SetBooks(Book[] books) {
    Assert.AssertEquals(1,
        books.Length);
    Assert.AssertEquals(1,
        books[0].Id);
} }
```

e) Offsite mock contributor

```
[Extension]
[Plug("Books")]
class OffsiteMock : Books {
  Book[] GetBooks() {
    return new Book[] {
        new MockBook(2) };
  }
  void SetBooks(Book[] books) {
    Assert.AssertEquals(1,
        books.Length);
    Assert.AssertEquals(2,
        books[0].Id);
} }

class MockBook : Book {
  MockBook(int id) { Id = id; }
  int Id { private set; get; }
  // ...
}
```

f) Functional test

```
class LibraryTest {
  MemoryStream stream = null;
  [Test(PluggedContributors = { "LocalMock", "OffsiteMock"} )]
  void Test(Library testee) {
    if (stream == null) {
      stream = new MemoryStream();
      testee.ExportBooks(stream);
    } else {
      stream.Position = 0;
      testee.ImportBooks(stream);
      stream = null;
} } }
```

1a

1b

Figure 80: Test case that finds a same on every run vs. unpredictable order fault (same contract)

```
[Extension]
[Plug("Application")]
[Slot("Books", OnPlugged = "AddBookStore",
    OnUnplugging = "RemoveBookStore")]
class Library : IApplication {
  List<Books> bookStores = new List<Books>();
  void AddBookStore(CompositionEventArgs args) {
    bookStores.add((Books) args.Plug.Extension.Object);
  }
  void RemoveBookStore(CompositionEventArgs args) {
    bookStores.remove((Books) args.Plug.Extension.Object);
  }
  void ExportBooks(Stream output) {
    var formatter = new BinaryFormatter();
    int size = bookStores.Count;
    formatter.Serialize(output, size);
    for (int store = 0; store < size; ++store) {
      Books b = bookStores[store];
      // ... export the books of the store ...
    }
  }
  void ImportBooks(Stream input) {
    var formatter = new BinaryFormatter();
    int size = (int) formatter.Deserialize(input);
    for (int store = 0; store < size; ++store) {
      Books b = bookStores[store];
      // ... import the books of the store ...
    }
  }
}
```

Figure 81: Library host with same on every run vs. unpredictable order fault (same contract)

5.2.2.2.4 Same order on every run vs. unpredictable order (different contracts)

The test case in Figure 82 tests the library host (cf. Figure 83) with a local book store contributor and a statistics tool. The test case plugs the contributors twice, each time in a different order. When the test case progresses to the second *plug LocalBooks* operation, it exposes the order fault in the library host. The cause for the fault is that the host misuses the *UseStatistics* setting, which causes a null pointer exception when the composition order of book store and statistics tool changes. A correct implementation of this host would access the statistics only if the *UseStatistics* setting is enabled and (*&&* instead of ||) the field *statistics* is not null. Furthermore, the host would not modify the *UseStatistics* setting at all, because this should only be done by the user.

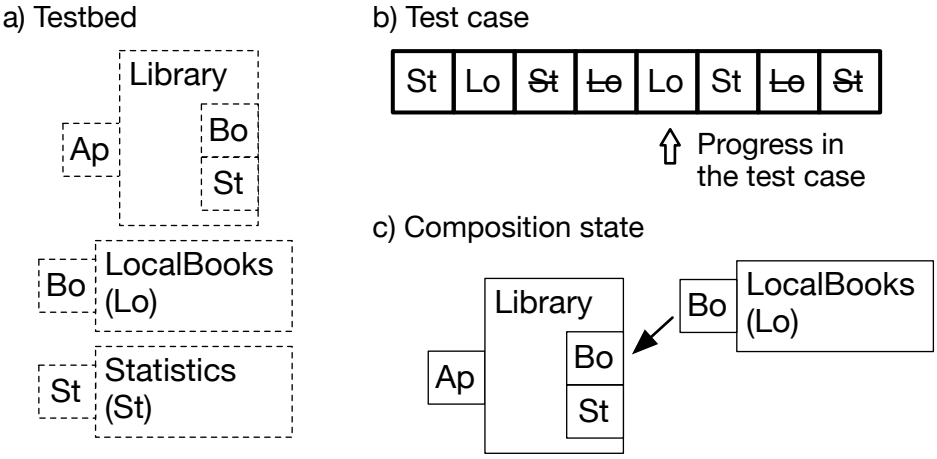


Figure 82: Test case that finds a same on every run vs. unpredictable order fault (different contracts)

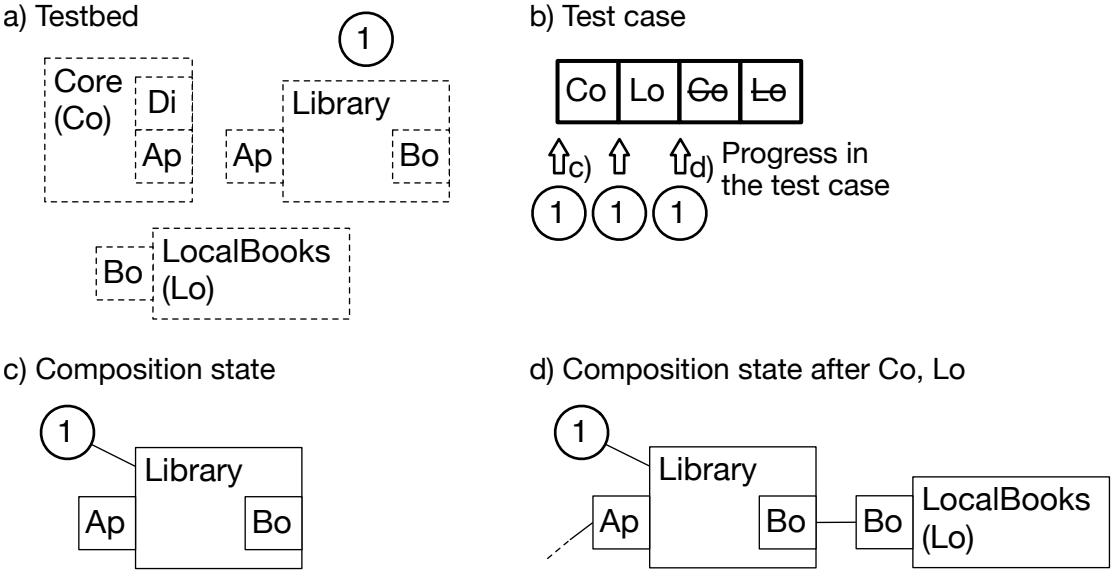
```
[Extension]
[Plug("Application")]
[Slot("Books", OnPlugged = "SetBookStore")]
[Slot("Statistics", OnPlugged = "SetStatistics",
      OnUnplugging = "RemoveStatistics")]
class Library : IApplication, IDisposable {
  Dictionary<String, bool> settings = LoadSettingsFromFile();
  Books bookStore;
  Statistics statistics;
  Dictionary<String, bool> LoadSettingsFromFile() { ... }
  void SaveSettingsToFile() { ... }
  void SetBookStore(CompositionEventArgs args) {
    bookStore = (Books) args.Plug.Extension.Object;
    if (settings.ContainsKey("UseStatistics")
        && settings["UseStatistics"] || statistics != null) {
      settings["UseStatistics"] = true;
      statistics.UpdateBookCount(bookStore.Count);
    } else {
      settings["UseStatistics"] = false;
    }
  }
  void SetStatistics(CompositionEventArgs args) {
    statistics = (Statistics) args.Plug.Extension.Object;
  }
  void RemoveStatistics(CompositionEventArgs args) {
    statistics = null;
  }
  void Dispose() {
    SaveSettingsToFile();
  }
}
```

Figure 83: Library host with same on every run vs. unpredictable order fault (different contracts)

5.2.2.2.5 All at once vs. continuously (same contract)

The test case in Figure 84 tests the library host (cf. Figure 85) with the local books store and a functional test, which is executed after each composition operation.

The functional test calls the *UpdateBooksOrdered* method in the library host, which retrieves data from its books contributors. When the test case progresses to the *plug LocalBooks* operation, it exposes the order fault in the library host. The cause for the fault is that the library host sets the length of the result array *booksOrdered* to the number of available contributors on the first call (zero contributors in our example, cf. Figure 84c). On subsequent calls it writes the data retrieved from the then available contributors into the array. If the number of contributors increases between calls (from zero to one, cf. Figure 84d), this causes an index out of bounds exception, because the result array is too small. A correct implementation of this host would resize the result array according to the number of available contributors on each call.



f) Functional test

```
[Test]
void Test(Library testee) {
  testee.UpdateBooksOrdered();
}
```

1

Figure 84: Test case that finds an all at once vs. continuously order fault (same contract)

```
[Extension]
[Plug("Application")]
[Slot("Books")]
class Library : IApplication {
  int[] booksOrdered = null;
  Extension self;
  public Library(Extension self) {
    this.self = self;
  }
  void UpdateBooksOrdered() {
    int count = self.Slots["Books"].PluggedPlugs.Count;
    if (booksOrdered == null) {
      booksOrdered = new int[count];
    }
    for (int index = 0; index < count; index++) {
      Plug p = self.Slots["Books"].PluggedPlugs[index];
      var books = (Books) p.Extension.Object;
      int nrBooksOrdered = 0;
      foreach (Book b in books.GetBooks()) {
        if (b.IsOrdered) {
          nrBooksOrdered++;
        }
      }
      booksOrdered[index] = nrBooksOrdered;
    }
  }
}
```

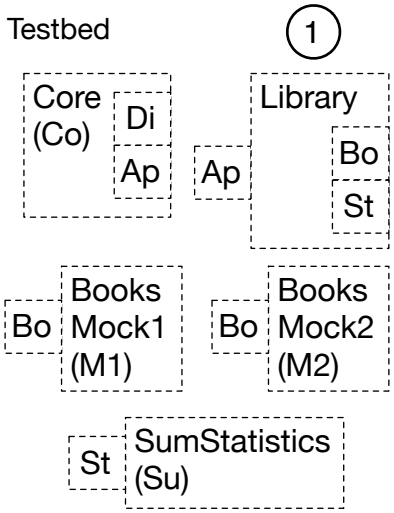
Figure 85: Library host with an all at once vs. continuously order fault (same contract)

5.2.2.2.6 All at once vs. continuously (different contracts)

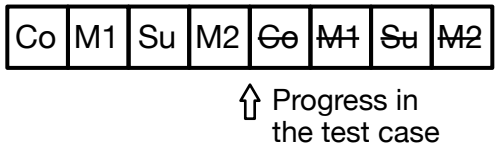
The test case in Figure 86 tests the library host (cf. Figure 87) with two book store mocks (cf. Figure 86e), a statistics contributor to calculate the sum of the book prices (cf. Figure 88), and a functional test. The functional test retrieves the sum of book prices calculated by the statistics contributor. The functional test is annotated such that it is only executed when the contributors *BooksMock1*, *BooksMock2*, and *Statistics* are plugged into the testee. In this composition each of the mocks has books with a total price of 6, which the statistics tool adds up to a total of 12. When the test case progresses to the plug *BooksMock2* operation, the functional test exposes the order fault in the library host. The cause for this fault is that the library host incorrectly stores the contributors in its own fields and that it disconnects the event handler *AddBookStore* when a statistic contributor is connected and disconnects the event handler *AddStatistics* when a book store is connected. Therefore in this test case the books of *BooksMock2* are ignored. Please note, that all book stores were composed before the statistics tool (e.g., M1, M2, Su) or the statistics tool was composed before the book stores (e.g., Su, M1, M2), the host would calculate the correct sum. Whenever book stores and the statistics tool are composed in an interleaved way (e.g., M1, Su, M2), the host calculates an incorrect sum. A correct implementation of this host would not rely

on the fact that all book stores (and statistics tools) become available at once (i.e., in a bulk), but instead would use the currently plugged book stores and statistics tools by retrieving them from the composition state on demand.

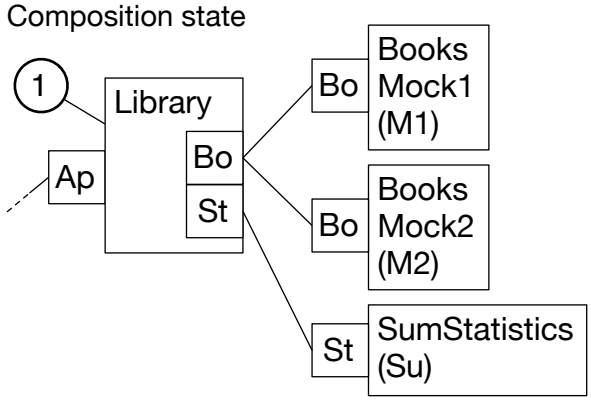
a) Testbed



b) Test case



c) Composition state



e) Mocks: BooksMock1, BooksMock2

```
[Extension("BooksMock1")]
...
[Extension("BooksMock2")]
[Plug("Books")]
class BooksMock : Books {
  Book[] GetBooks() {
    return new Book[] {new MockBook(2), new MockBook(4)};
  }
  // ...
}
class MockBook : Book {
  MockBook(int price) { Price = price; }
  int Price { private set; get; }
  // ...
}
```

f) Functional test

```
      [Test(PluggedContributors = { "BooksMock1", "BooksMock2",
                                    "Statistics"} )]
(1)   void Test(Library testee) {
        Assert.AssertEquals(12, testee.GetStatistics("sum"));
      }
```

Figure 86: Test case that finds an all at once vs. continuously order fault (different contracts)

```
[Extension]
[Plug("Application")]
[Slot("Books")]
[Slot("Statistics")]
class Library : IApplication {
  Extension self;
  List<Books> bookStores = new List<Books>();
  Dictionary<String, Statistics> statisticsMap =
      new Dictionary<String, Statistics>();
  Library(Extension self) {
    this.self = self;
    self.Slots["Books"].Plugged += AddBookStore;
    self.Slots["Statistics"].Plugged += AddStatistics;
  }
  int GetStatistics(String name) {
    if (!statisticsMap.ContainsKey(name)) { return -1; }
    Statistics statistics = statisticsMap[name];
    List<Book> allBooks = new List<Book>();
    foreach (Books books in bookStores) {
      allBooks.AddRange(books.GetBooks());
    }
    return statistics.Calculate(allBooks);
  }
  void AddBookStore(CompositionEventArgs args) {
    if (statisticsMap.Count > 0) {
      self.Slots["Statistics"].Plugged -= AddStatistics;
    }
    bookStores.Add((Books) args.Plug.Extension.Object);
  }
  void AddStatistics(CompositionEventArgs args) {
    if (bookStores.Count > 0) {
      self.Slots["Books"].Plugged -= AddBookStore;
    }
    var statistics = (Statistics) args.Plug.Extension.Object;
    var operation = (String) args.Plug.Params["Operation"].Value;
    statisticsMap[operation] = statistics;
  }
}
```

Figure 87: Library host with all at once vs. continuously order fault (different contracts)

```
[Extension]
[Plug("Statistics")]
[Param("Operation", "sum")]
class SumStatistics : Statistics {
  int Calculate(List<Book> books) {
    int sum = 0;
    foreach (Book b in books) {
      sum += b.Price;
    }
    return sum;
  }
}
```

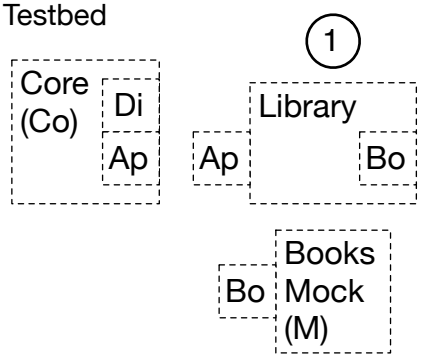
Figure 88: Statistics contributor that calculates the total price of books

5.2.2.3 Duration faults

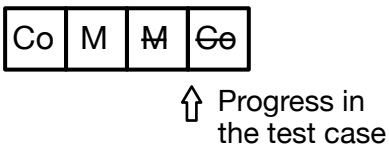
A duration fault can be found by limiting the duration of a contributor's availability and performing a functional test. The functional test must be executed after a contributor was disconnected.

The test case in Figure 89 composes the library host (cf. Figure 90) with a books mock and a functional test, which calls the *GetBook* method where the library host accesses the books contributor. When the test case progresses to the composition operation *unplug Mock*, the functional test is executed and exposes the duration fault in the library host. The cause for the fault is that the host tries to use the books contributor after it was removed, which causes the assertion in *BooksMock* to fail, as it is already disposed. A correct implementation of this host would listen to the *Unplugged* event and set *bookStore* to *null* in response to it.

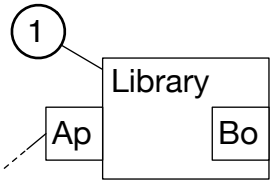
a) Testbed



b) Test case



c) Composition state



e) Mock contributor

```
[Extension]
[Plug("Books")]
class BooksMock : Books, IDisposable {
  bool isDisposed = false;
  Book GetBook(int bookId) {
    Assert.AssertFalse(isDisposed);
    return null;
  }
  void Dispose() { isDisposed = true; }
}
```

f) Functional test

```
[Test]
void Test(Library testee) {
  testee.GetBook(1);
}
```

(1)

Figure 89: Test case that finds a permanent vs. temporary availability duration fault

```
[Extension]
[Plug("Application")]
[Slot("Books", OnPlugged = "SetBookStore")]
class Library : IApplication {
  Books bookStore;
  void SetBookStore(CompositionEventArgs args) {
    bookStore = (Books) args.Plug.Extension.Object;
  }
  Book GetBook(int bookId) {
    if (bookStore != null) {
      return bookStore.GetBook(bookId);
    }
    // ...
  }
}
```

Figure 90: Library host with permanent vs. temporary availability duration fault

5.2.3 Contributor identification faults

A contributor identification fault is a mismatch between how a host identifies its contributors, and how the composition mechanism identifies them. A contributor identification fault can be found by varying the contributors which are available. The test case in Figure 91 composes the library host (cf. Figure 92) with a books mock contributor. When the test case progresses to the *plug BooksMock* operation it exposes the contributor identification fault. The cause for the fault is that the host assigns the books mock to a *LocalBooks* field. A correct implementation of this host would treat the contributor as a component of type *Books* instead of type *LocalBooks*.

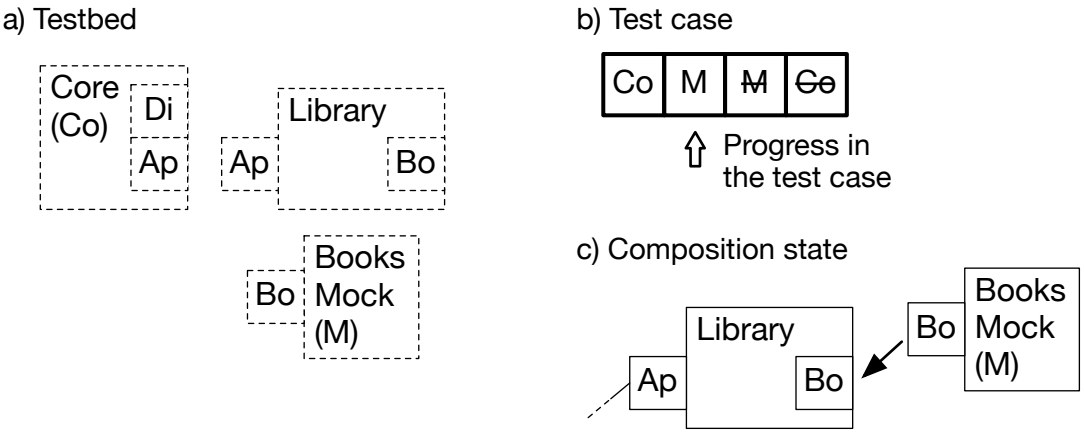


Figure 91: Test case that finds a contributor identification fault

```
[Extension]
[Plug("Application")]
[Slot("Books", OnPlugged = "AddBookStore")]
class Library : IApplication {
  LocalBooks books;
  void AddBookStore(CompositionEventArgs args) {
    books = (LocalBooks) args.Plug.Extension.Object;
    // ...
  }
}
```

Figure 92: Library host with a contributor identification fault

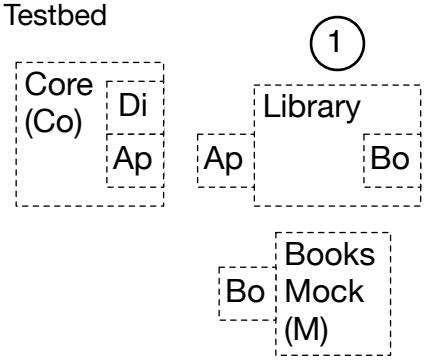
5.2.4 Contributor instantiation faults

A contributor instantiation fault is a mismatch in terms of who creates a contributor (i.e., the host itself or the composition mechanism), or how contributors are created (i.e., uniformly for every host or in a host-specific way). A contributor instantiation fault can be found by doing a functional test that checks whether the testee actually uses a contributor instance provided by Plux; a faulty testee would use a self-created instance instead. Moreover, the functional test should check if the testee behaves correctly if the contributor is shared with other hosts.

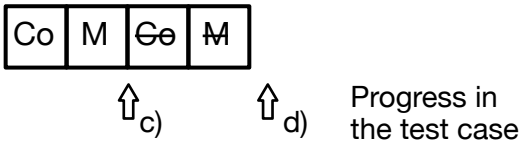
5.2.4.1 By host vs. by infrastructure

The test case in Figure 93 composes the library host (cf. Figure 94) with a books mock and a functional test. The functional test calls the *GetBook* method of the library host, which accesses the books contributor. Later when the test case progresses to *unplug BooksMock*, it exposes the instantiation fault in the library host. The cause for the fault is that the host does not use the books mock instance plugged by the infrastructure, but creates another instance itself. Thus the *GetBook* call of the functional test does not reach the books mock, which breaks the assertion in the mock when it is unplugged. Furthermore, if the corresponding Act runtime check for detecting composition standard violations is installed, it raises a run-time error already when the library host calls the self-created instance of the books mock (cf. Figure 93c), because a host must not call contributors that are not plugged into it. A correct implementation of this host would use the books contributor provided in the *Plugged* event.

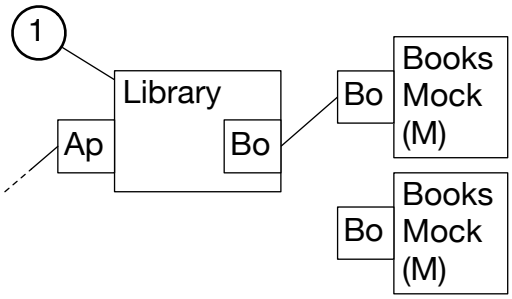
a) Testbed



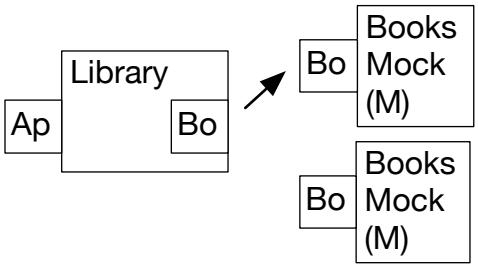
b) Test case



c) Composition state after Co, M



d) Composition state after Co, M, ~~Co~~, ~~M~~



e) Mock contributor

```
[Extension]
[Plug("Books", OnUnplugged = "CheckIsUsed")]
class BooksMock : Books {
  bool isUsed;
  Book GetBook(int bookId) { isUsed = true; return null; }
  void CheckIsUsed(CompositionEventArgs args) {
    Assert.AssertTrue(isUsed);
  }
}
```

f) Functional test


```
[Test]
void Test(Library testee) {
  testee.GetBook(1);
}
```

Figure 93: Test case that finds a by host vs. by infrastructure instantiation fault

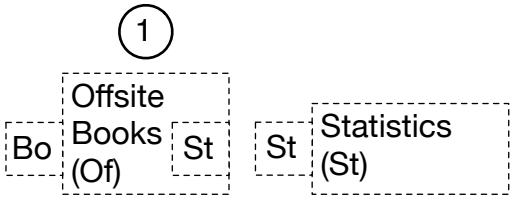
```
[Extension]
[Plug("Application")]
[Slot("Books", OnPlugged = "AddBookStore")]
class Library : IApplication {
  List<Books> bookStores = new List<Books>();
  Composer composer;
  Library (Extension self) {
    composer = self.Composer;
  }
  void AddBookStore(CompositionEventArgs args) {
    Extension extension = composer.Create(
        args.Plug.Extension.ExtensionType);
    bookStores.Add((Books) extension.Object);
  }
  Book GetBook(int bookId) {
    foreach (Books bookStore in bookStores) {
      Book b = bookStore.GetBook(bookId);
      if (b != null)  {
        return b;
      }
    }
    return null;
  }
}
```

Figure 94: Library host with a by host vs. by infrastructure instantiation fault

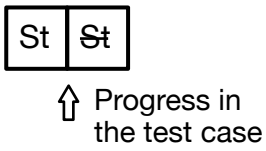
5.2.4.2 Globally uniform vs. host-specific

The test case in Figure 95 composes the offsite book store host (cf. Figure 96) with a statistics contributor and a functional test. The functional test compares the total price of the books in the offsite book store provided by the book store with the price provided by the statistics tool for this book store. To simulate that the statistics tool is also used by another host, the test case adds a dummy book from some other book store to the statistics. Please note, this simulation corresponds to a composition where some other host retrieves the statistics contributor from the composition state in order to share it (cf. Section 3.3.4.2 on page 57). When the test case progresses to the composition operation *plug Statistics* the functional test exposes the instantiation fault in the book store host. The cause for the fault is that the host uses the method *GetTotalPrice()* that returns the total price for all books in the statistics instead of *GetTotalPrice(int bookStoreId)* that returns the total price for the books of a given book store.

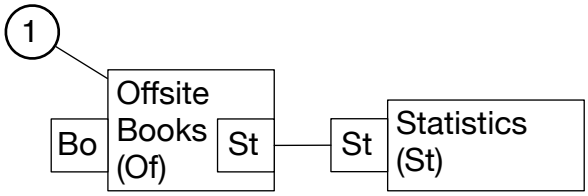
a) Testbed



b) Test case



c) Composition state



d) Functional test

1

```
[Test(PluggedContributors = { "Statistics"} )]
void Test(Extension testee) {
  var books = (OffsiteBooks) testee.Object;
  var statistics = (Statistics) testee.Slots["Statistics"]
      .PluggedPlugs[0].Extension.Object;
  int dummyBooksId = books.Id + 1;
  statistics.Add(dummyBooksId, new MockBook(1000));
  int booksValue = books.GetTotalPrice();
  int statisticValue = statistics.GetTotalPrice(books.Id);
  Assert.AssertEquals(booksValue, statisticValue);
}

class MockBook : Book {
  MockBook(int price) { Price = price; }
  int Price { private set; get; }
  // ...
}
```

Figure 95: Test case that finds a globally uniform vs. host-specific instantiation fault

```
[Extension]
[Plug("Books")]
[Slot("Statistics")]
class OffsiteBooks : Books {
  Extension self;
  int id = ...;
  OffsiteBooks(Extension self) {
    this.self = self;
  }
  int GetTotalPrice() {
    var stat = (Statistics) self.Slots["Statistics"]
        .PluggedPlugs[0].Extension.Object;
    foreach (Book book in ...) {
      stat.Add(id, book);
    }
    return stat.GetTotalPrice();
  }
  int Id { get { return id; } }
}

[Extension]
[Plug("Statistics")]
class StatisticsTool : Statistics {
  void Add(int bookStoreId, Book book) { ... }
  // Get the total price for the books in the given book store
  int GetTotalPrice(int bookStoreId) { ... }
  // Get the total price for all books
  int GetTotalPrice() { ... }
}
```

Figure 96: OffsiteBooks host with a globally uniform vs. host-specific instantiation fault

5.2.5 Contributor registration faults

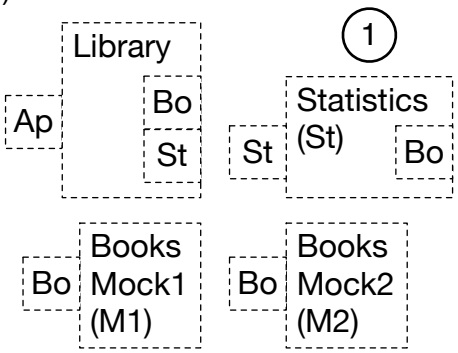
A contributor registration fault is a mismatch in terms of where a composition mechanism makes contributors available (i.e., globally to all hosts or specifically to individual hosts) or how a composition mechanism tracks contributor usage (i.e., by storing just a global usage counter per contributor or by keeping track of which hosts are connected to which contributors). A contributor registration fault can occur if contributors are used by multiple hosts. It can be found if some contributors are connected to one set of hosts, whereas other contributors are connected to another set of hosts.

The test case in Figure 97 composes the library host (cf. Figure 98) with a statistics contributor, two book store mocks, and a functional test. The functional test calls the method *CalculateAveragePrice* of the library host and compares the result to the expected value. When the test case progresses to the composition operation *plug BooksMock1 into Library*, it exposes the registration fault in the library host, because the library uses just one book store whereas the statistics tool is connected to two book stores. The cause for the fault is that the host uses the method *GetTotalPrice()* that returns the total price for all book stores in the

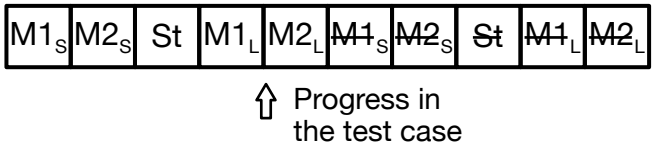
statistics, instead of *GetTotalPrice(int bookStoreId)* to calculate the price only for the book store that is plugged into the library.

Please note, that this example covers the contributor registration fault *global-usage vs. host-specific usage* (cf. Section 3.3.5.1 on page 60). The contributor registration fault *global availability vs. host-specific availability* (cf. Section 3.3.5.2 on page 61) does not occur in Plux. However, if this test case is applied to hosts that use a composition mechanism where this fault can occur, the test case would find the fault, because for the fault it is irrelevant if the host could but does not use (usage) or if it cannot use the contributor, because it is not available (availability).

a) Testbed



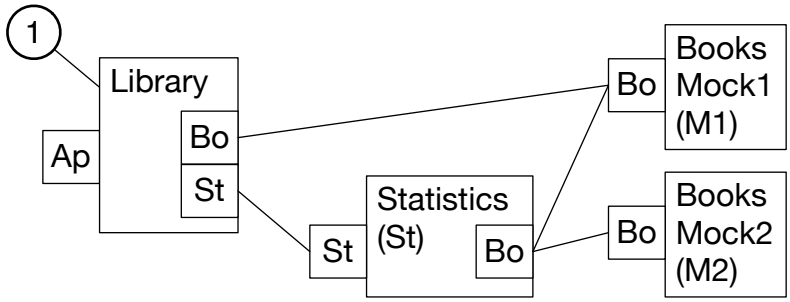
b) Test case



X_S.. on host Statistics

X_L .. on host Library

c) Composition state



d) Mocks: BooksMock1, BooksMock2

```
[Extension("BooksMock1")]
[Plug("Books")]
class BooksMock1 : Books {
  Book[] GetBooks() {
    return new Book[] {
        new MockBook(2),
        new MockBook(4)};
  }
  // ...
}
```

```
[Extension("BooksMock2")]
[Plug("Books")]
class BooksMock2 : Books {
  Book[] GetBooks() {
    return new Book[] {
        new MockBook(10)};
  }
  // ...
}

class MockBook : Book {
  MockBook(int price) { Price = price; }
  int Price { private set; get; }
  // ...
}
```

e) Functional test

```
[Test(PluggedContributors = { "BooksMock1" } )]
void Test(Library testee) {
  Assert.AssertEquals(3, testee.CalculateAveragePrice());
}
```

1

Figure 97: Test case that finds a global vs. host-specific registration fault

```
[Extension]
[Plug("Application")]
[Slot("Books")]
[Slot("Statistics")]
class Library : IApplication {
  ...
  int CalculateAverageValue() {
    Slot s = self.Slot("Books");
    int count = 0;
    foreach (Plug p in s.PluggedPlugs) {
      var books = (Books) p.Extension.Object;
      count += books.Count;
    }
    int totalPrice = 0;
    if (self.Slots["Statistics"].Length != 0) {
      var stat = (Statistics) self.Slots["Statistics"]
          .PluggedPlugs[0].Extension.Object;
      totalPrice = stat.GetTotalPrice();
    } else {
      foreach (Plug p in s.PluggedPlugs) {
        var books = (Books) p.Extension.Object;
        foreach (Book b in books.GetBooks()) {
          totalPrice += b.Price;
        }
      }
    }
    return totalPrice / count;
  }
}
[Extension]
[Plug("Statistics")]
[Slot("Books")]
class StatisticsTool : Statistics {
  int GetTotalPrice(int bookStoreId) { ... }
  int GetTotalPrice() { ... }
}
```

Figure 98: Library host with a global vs. host-specific registration fault

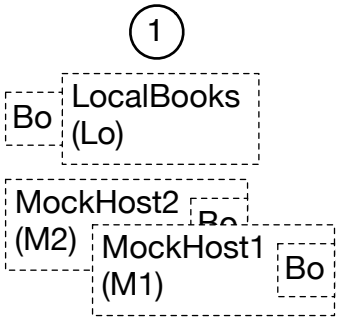
5.2.6 Contributor sharing faults

A sharing fault in a contributor causes errors in a correctly implemented host, if a contributor that does not support sharing is shared among hosts by the composition mechanism. Thus a sharing fault in a contributor can be found if the contributor is indeed shared, typically after the contributor was disconnected from some hosts and is still connected to other hosts.

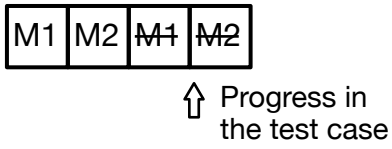
The test case in Figure 99 composes the local book store contributor (cf. Figure 100) with two mock hosts and a functional test. The functional test calls the method *GetBooks* of the book store. When the test case progresses to the composition operation *unplug MockHost1*, the functional test exposes the contributor sharing fault in the book store contributor, because the *GetBooks* method causes a null pointer exception when it passes the null reference from the field *connec-*

tion into the *SqlCommand* constructor. The cause for the fault is that the local book store sets its field *connection* to *null* when it is unplugged from mock host 1. A correctly implemented contributor would only close the connection when it is unplugged from the last host (*if (args.Plug.SlotsWherePlugged.Count == 0) { ... }*).

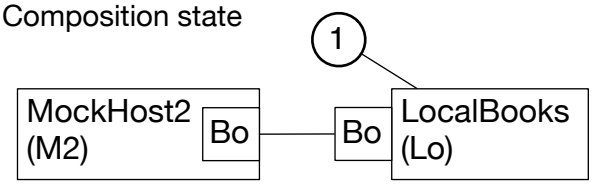
a) Testbed



b) Test case



c) Composition state



d) Mocks: MockHost1, MockHost2

```
[Extension]                [Extension]
[Slot("Books")]            [Slot("Books")]
class MockHost1 { }        class MockHost2 { }
```

e) Functional test

```
    [Test(UsedPlugs = { "Books" } )]
1   void Test(LocalBooks testee) {
      testee.GetBooks("Myers");
    }
```

Figure 99: Test case that finds a contributor sharing fault

```
[Extension]
[Plug("Books", OnPlugged="OpenDatabase",
      OnUnplugging="CloseDatabase")]
class LocalBooks : Books {
  SqlConnection connection;
  void OpenDatabase(CompositionEventArgs args) {
    connection = new SqlConnection(...);
    connection.Open();
  }
  void CloseDatabase(CompositionEventArgs args) {
    connection.Close();
    connection = null;
  }
  Book[] GetBooks(String author) {
    SqlCommand command =
        new SqlCommand("SELECT * ...", connection);
    SqlDataReader reader = command.ExecuteReader();
    ...
  }
}
```

Figure 100: Book store contributor with a contributor sharing fault

5.2.7 Composition standard violations

The composition standard specifies the rules according to which a composition must be done in order to be valid. A composition standard violation can be found by a functional test that simply calls a method on the testee. Assertions are unnecessary, because Act raises a runtime error during the method call if a composition standard is violated. The composition standard violation *composition state mismatch* does not cause a runtime error and must thus be tested using mock contributors. The mock contributors check if they were actually used after the functional test has been executed. A faulty testee does not use all mock contributors.

5.2.7.1 Continued use of an unplugged contributor

The test case in Figure 101 composes the library host (cf. Figure 102) with a local book store and a functional test, which calls the method *GetBooks* of the library host. When the test case progresses to the composition operation *unplug LocalBooks*, the functional test exposes the *continued use of unplugged contributor* fault, because the *GetBooks* call to the local books contributor causes a Plux runtime error. The cause for the fault is that the host tries to call the *GetBooks* method on the local books contributor, although this contributor is unplugged. A correct implementation of this host would react to the *Unplugged* event and would update the book store collection accordingly.

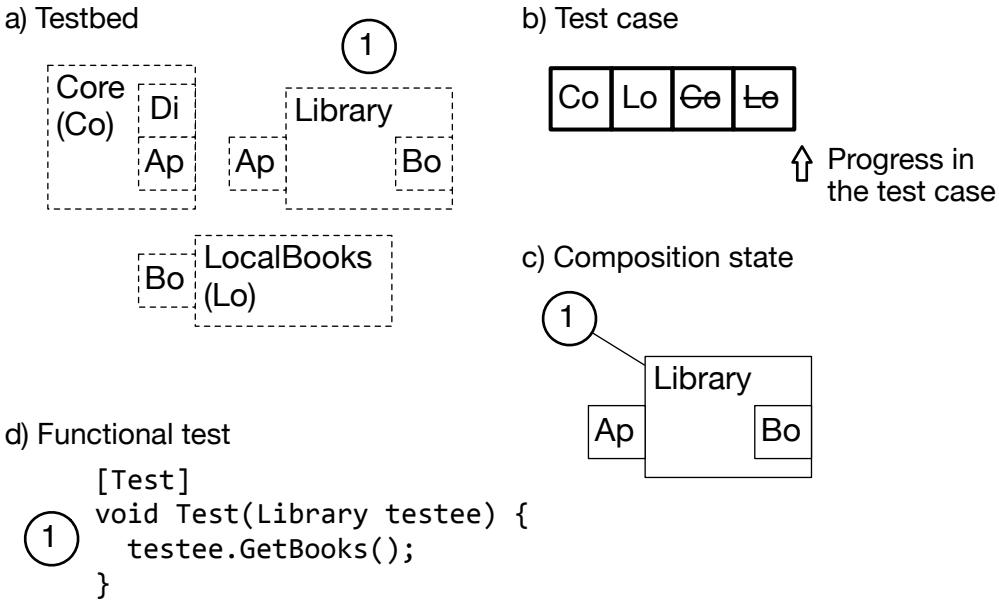


Figure 101: Test case that finds a use-of-unplugged-contributor fault

```
[Extension]
[Plug("Application")]
[Slot("Books", OnPlugged = "AddBookStore")]
class Library : IApplication {
  List<Books> bookStores = new List<Books>();
  void AddBookStore(CompositionEventArgs args) {
    bookStores.Add((Books) args.Plug.Extension.Object);
  }
  Book[] GetBooks() {
    List<Book> books = new List<Book>();
    foreach (Books b in bookStores) {
      books.AddRange(b.GetBooks());
    }
    return books.ToArray();
  }
}
```

Figure 102: Library host with a use-of-an-unplugged-contributor fault

5.2.7.2 Use of not-plugged component

The test case in Figure 103 composes the library host (cf. Figure 104) with a local book store and a functional test, which calls the method *GetBooks* of the library host. When the test case progresses to the composition operation *plug Local-Books*, the functional test exposes the *use of not-plugged component* fault, because the *GetBooks* call to the local book store contributor instance that is not plugged into the library causes a Plux runtime error. The cause for the fault is that the host uses its self-created instance without plugging it into its slot. A correct implementation of this host would plug the self-created instance before using it. However, a better solution would be to use the contributors which were plugged into the *Books* slot by Plux.

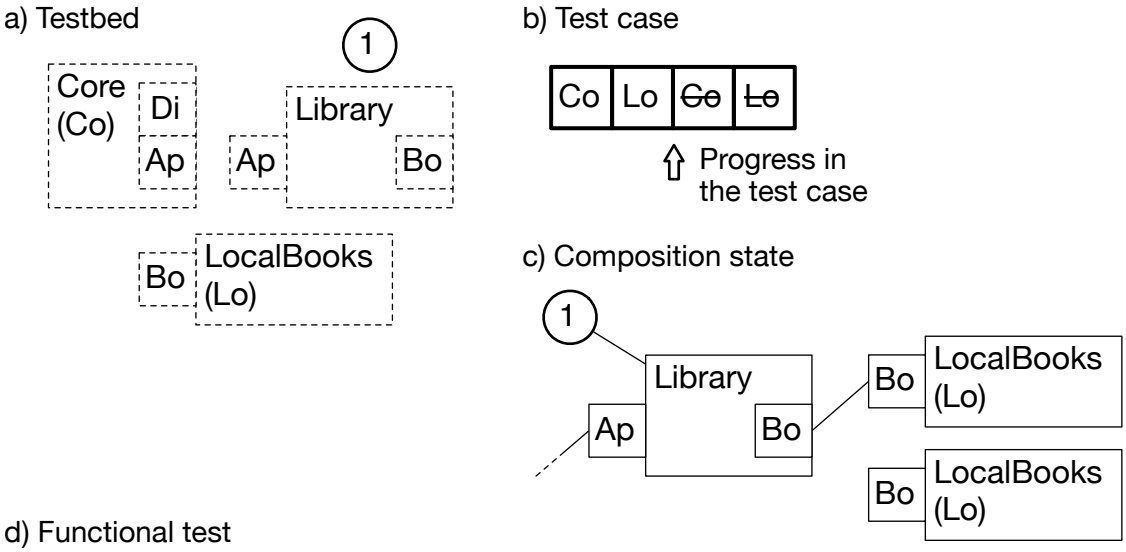


Figure 103: Test case that finds a use-of-a-not-plugged-component fault

```
[Extension]
[Plug("Application")]
[Slot("Books")]
class Library : IApplication {
  Composer composer;
  TypeStore typeStore;
  Library(Extension self) {
    Runtime runtime = self.Runtime;
    composer = runtime.Composer;
    typeStore = runtime.TypeStore;
  }
  Book[] GetBooks() {
    List<Book> books = new List<Book>();
    foreach (PlugType pt in typeStore.GetPlugTypes("Books")) {
      Plug p = composer.Create(pt.ExtensionType).Plugs["Books"];
      Books b = (Books) p.Extension.Object;
      books.AddRange(b.GetBooks());
    }
    return books.ToArray();
  }
}
```

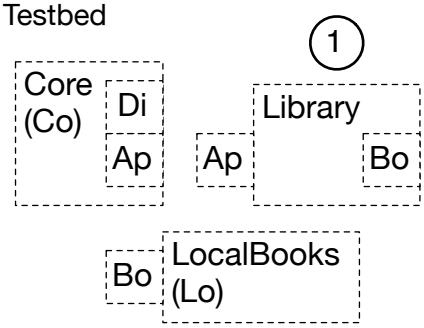
Figure 104: Library host with a use-of-a-not-plugged-component fault

5.2.7.3 Contributor call in non-runtime thread

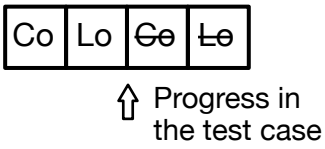
The test case in Figure 105 composes the library host (cf. Figure 106) with a local book store and a functional test, which calls the method *GetTotalPrice* of the library host. When the test case progresses to the composition operation *plug LocalBooks*, the functional test exposes the *contributor call in non-runtime thread* fault, because *GetTotalPrice* uses worker threads, which call a method on the at-

tached local book stores. The worker threads call *CalculateTotalPrice* and finally *GetBooks* on the given book store. Plux forbids contributor calls from outside the Plux runtime thread. A correct implementation of this host would retrieve the books from the contributors in the runtime thread and pass them to the worker threads.

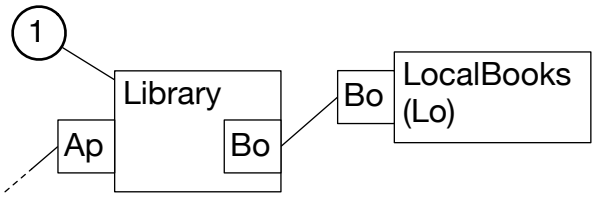
a) Testbed



b) Test case



c) Composition state



d) Functional test

```
[Test(PluggedContributors = { "LocalBooks" })]
void Test(Library testee) {
  Assert.AssertGreaterOrEqual(0, testee.GetTotalPrice());
}
```

1

Figure 105: Test case that finds a contributor-call-in-non-runtime thread fault

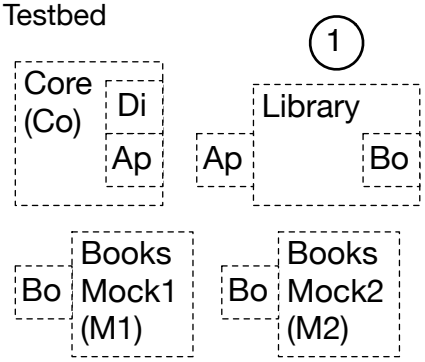
```
[Extension]
[Plug("Application")]
[Slot("Books")]
class Library : IApplication {
  int GetTotalPrice() {
    Slot s = self.Slots["Books"];
    List<Task<int>> tasks = new List<Task<int>>();
    foreach (Plug p in s.PluggedPlugs) {
      var books = (Books) p.Extension.Object;
      Func<int> func = delegate() {
          return CalculateTotalPrice(books);
      };
      Task<int> task = Task<int>.Factory.StartNew(func);
      tasks.Add(task);
    }
    int totalPrice = 0;
    foreach (Task<int> task in tasks) {
      totalPrice += task.Result;
    }
    return totalPrice;
  }
  int CalculateTotalPrice(Books books) {
    int bookStorePrice = 0;
    foreach (Book b in books.GetBooks()) {
      bookStorePrice += b.Price;
    }
    return bookStorePrice;
  }
}
```

Figure 106: Library host with a contributor-call-in-non-runtime-thread fault

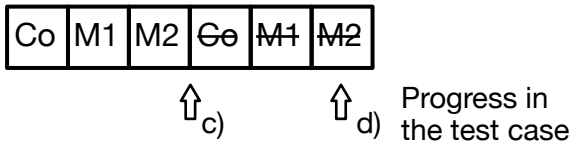
5.2.7.4 Composition state mismatch

The test case in Figure 107 composes the library host (cf. Figure 108) with two book store mocks and a functional test, which calls the *GetBooks* method of the library host. When the test case progresses to the composition operation *plug BooksMock2*, the functional test is executed to trigger the use of the book store mocks. When the test case progresses to the composition operation *unplug BooksMock2*, it exposes the *composition state mismatch* fault, because the assertion in the book store mock fails. The cause for this fault is that the library host only uses the first plugged contributor and ignores the others. As this host actually requires only one contributor, a correct implementation would make sure that Plux connects only one contributor by applying a behavior.

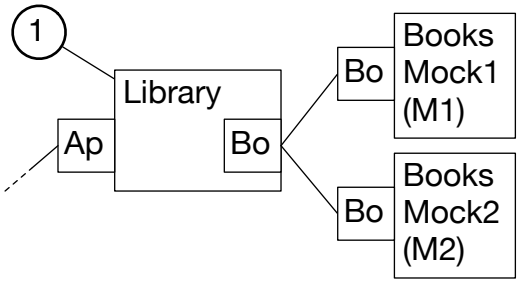
a) Testbed



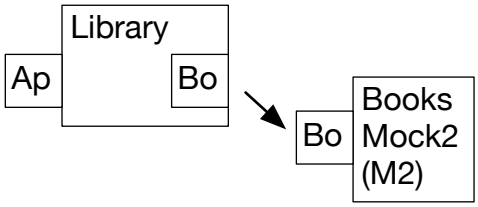
b) Test case



c) Composition state after Co, M1, M2



d) Composition state after Co, M1, M2, ~~Co~~, ~~M1~~, ~~M2~~



e) Mock contributor

```
[Extension]
[Plug("Books", OnUnplugged = "CheckIsUsed")]
class BooksMock : Books {
  bool isUsed;
  Book[] GetBooks() {
    isUsed = true;
    return new Book[] { new MockBook(10) };
  }
  void CheckIsUsed(CompositionEventArgs args) {
    Assert.AssertTrue(isUsed);
  }
}
```

f) Functional test


```
[Test(PluggedHosts = { "BooksMock1", "BookMocks2" })]
void Test(Library testee) {
  testee.GetBooks();
}
```

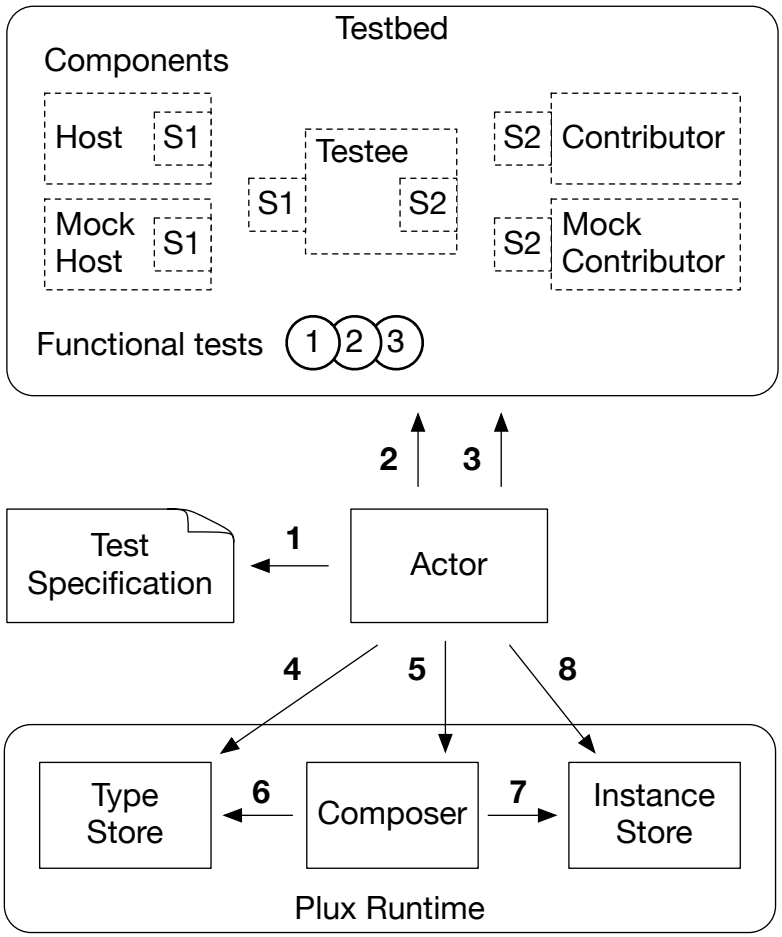
Figure 107: Test case that finds a composition state mismatch

```
[Extension]
[Plug("Application")]
[Slot("Books", OnPlugged = "SetBookStore",
      OnUnplugging = "ClearBookStore")]
class Library : IApplication {
  Books bookStore;
  void SetBookStore(CompositionEventArgs args) {
    if (bookStore == null) {
      bookStore = (Books) args.Plug.Extension.Object;
    }
  }
  void ClearBookStore(CompositionEventArgs args) {
    if (bookStore == args.Plug.Extension.Object) {
      bookStore = null;
    }
  }
  Book[] GetBooks() {
    return bookStore.GetBooks();
  }
}
```

Figure 108: Library host with a composition state mismatch

5.3 The automated composability test tool Actor

The automated composability test tool *Actor* implements the composability test method from Section 5.1. It generates the test cases for a given testbed, executes the test cases and the functional tests, and collects the tests results. Figure 109 shows how Actor uses the Plux runtime to execute the test cases: (1) it reads the test specification from a configuration file; (2) it prepares factories to create the testbed components; (3) it looks up the test assemblies required for the functional tests; (4) it discovers the testbed components, i.e., it inserts the components' metadata into the type store; (5) it uses the composer to execute the composition operations and wires proxies between extensions (the proxies are used to detect composition standard violations), thereby (6) the composer retrieves the metadata of the testbed components from the type store and (7) stores the instances and their connections into the instance store; (8) Actor executes the functional tests on the instances retrieved from the instance store. To improve performance, Actor executes the test cases concurrently, i.e., it performs the steps 4 to 8 for multiple test cases in parallel.



1 Read test specification
2 Create factories for testbed components
3 Look up test methods for functional tests
4 Discover testbed components
5 Execute composition operations and wire proxies between extensions
6 Get extension types
7 Store instance metadata and relationships
8 Execute functional tests on testee

Figure 109: Automated composability test tool architecture

Figure 110 shows the XML configuration file for the test specification in Figure 111, which is the configuration for the test case shown in Section 5.2.2.2.5 All at once vs. continuously (same contract). The testee is the *Library* extension in the *Library* plugin, the *Plux Core* with the *Application* slot is a mock host, the *LocalBooks* extension is a mock contributor, and the *FunctionalTest1* is a functional test. The composition scenarios in which the functional test should be executed are specified using the *UsedPlugs* and *UsedSlots* parameters of the *Test* attribute. In this example, both parameters are empty because the functional test should be executed in all composition scenarios. Please note, as this is the default, the parameters could be omitted. See Section 5.1.4 for the descriptions of the attributes

used in the Xml file and the parameters of the *Test* attribute. All assemblies must be located in one of the paths specified. As the testbed comprises two mocks (which in this case are actual components), the number of composition operations per test case is four. With four composition operations, the number of generated test cases is small enough. Thus a reduction of generated test cases is unnecessary and the tuple length is set to four.

```
<test-case tuplelength="4">
  <testee extension="Library" plugin="Library" />
  <mocks>
    <mock role="Contributor" extension="LocalBooks"
        plugin="LocalBooks" target="Library"
        usecount="1" count="1"
        plug="Books" slot="" />
    <mock role="Host" extension="Core"
        plugin="Plux" target="Library"
        usecount="1" count="1" plug=""
        slot="Application" />
  </mocks>
  <tests>
    <test class="FunctionalTest1" assembly="FunctionalTests" />
  </tests>
  <paths>
    <path name="Contracts"/> <path name="Plugins"/>
    <path name="Mocks"/> <path name="Functional Tests"/>
  </paths>
</test-case>
```

Figure 110: XML configuration file for the automated composability test tool

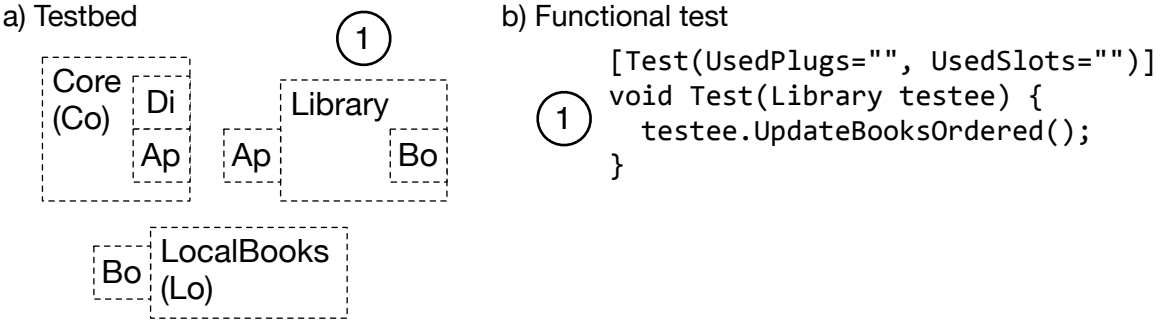


Figure 111: Example testbed for the automated composability test tool

The command *Actor Library.xml* invokes Actor with this configuration file. Figure 112 shows the output of Actor: on the left-hand side, the six generated test case with 4-tuples; on the right-hand side, the results of test case 1, with a failed assertion from functional test 1 and a stack trace that shows the source code location of the failed assertion. Please note, the thesis uses simplified drawn user interfaces of Actor, however, a real screenshot is given in the Appendix, see Figure 142 on page 154.

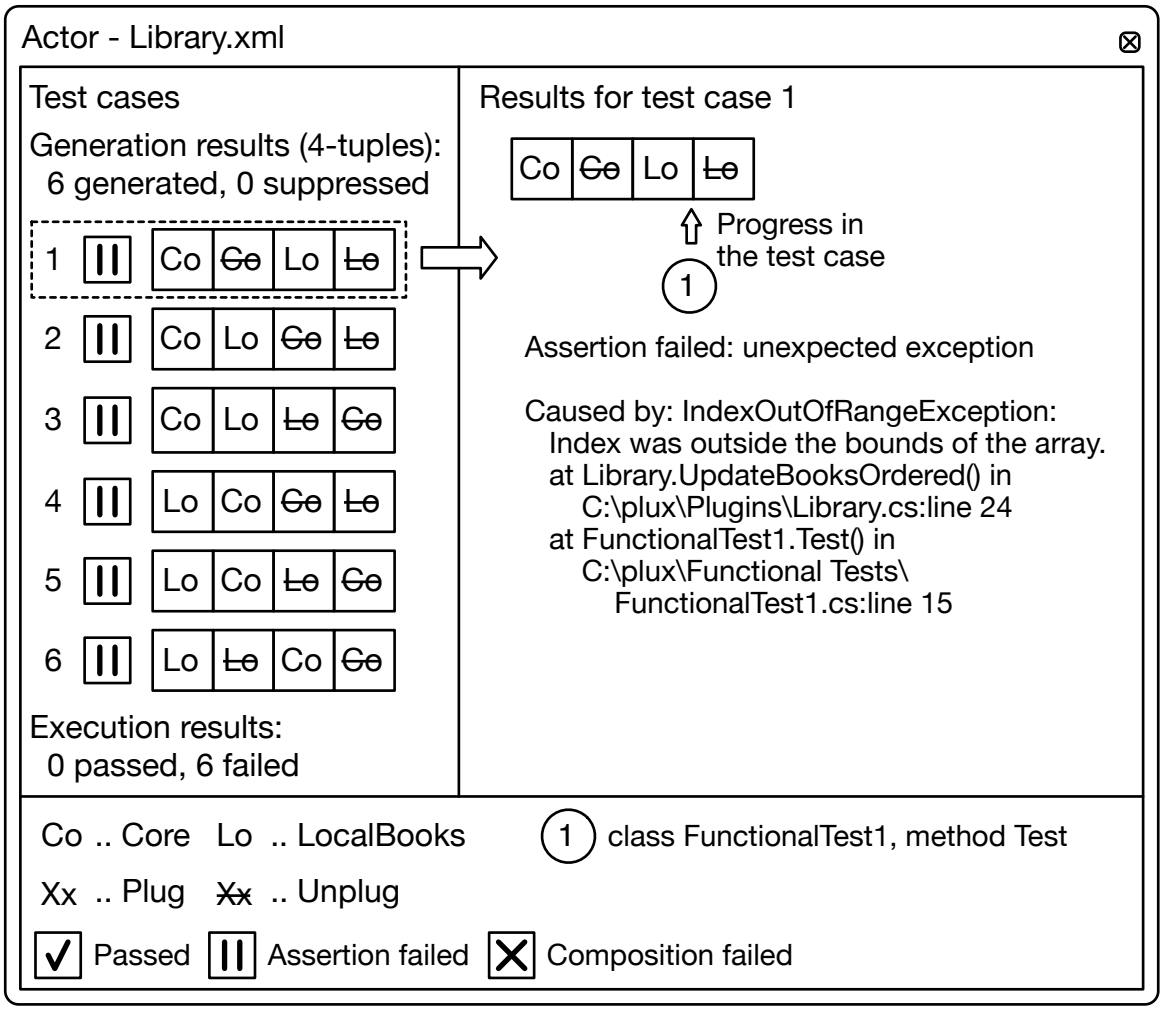


Figure 112: Test cases and results in the automated composability test tool

5.4 Experimental evaluation

We evaluated our testing approach in an experiment. The goal of study our is to compare composability testing with and without Actor in order to evaluate the automated composability test method. The quality focus is the effectiveness and efficiency of the automated composability test method. The experiment was performed by computer science students testing a Plux extension.

5.4.1 Experiment definition

The students had to test a Fibonacci extension, which is part of a calculator application. The extension calculates the Fibonacci number for a given number using contributors for the operations add, subtract, and exclusive or (cf. Figure 113).

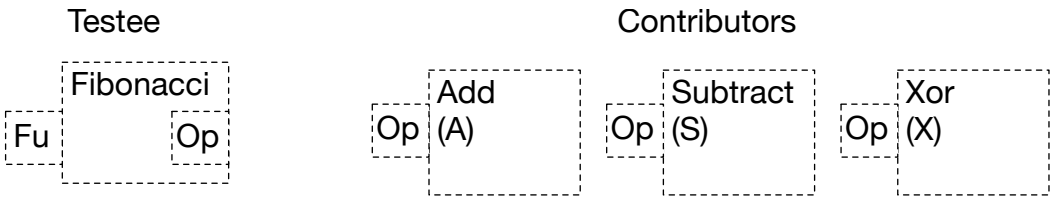


Figure 113: Testee and contributors used in our experimental evaluation

The Fibonacci extension is a contributor for the *Function* slot of the calculator application (not shown). Such contributors evaluate a function value for a given operand. The Fibonacci extension is also a host for contributors of the *Operation* slot (cf. Figure 114). A contributor for the *Operation* slot specifies its operation as a string using the *Symbol* parameter of the slot and calculates the result of the operation in the *Calculate* method. Figure 115 shows the testee and Figure 116 the contributors.

```
[SlotDefinition("Function")]
interface Function {
  // throws an InvalidOperationException if it cannot evaluate
  // because required contributors are unavailable
  int Evaluate(int operand);
}

[SlotDefinition("Operation")]
[ParamDefinition("Symbol", typeof(string))]
interface Operation {
  int Calculate(int operand1, int operand2);
}
```

Figure 114: Slot definitions used in experimental evaluation

```
[Extension]
[Plug("Function")]
[Slot("Operation", OnPlugged = "StoreContributor",
      OnUnplugging = "DisposeContributor")]
class FibOperation : Function {
  void StoreContributor(CompositionEventArgs args) { ... }
  void DisposeContributor(CompositionEventArgs args) { ... }
  int Evaluate(int operand) { ... }
}
```

Figure 115: Testee used in experimental evaluation

```
[Extension]
[Plug("Operation")]
[Param("Symbol", "+")]
class Add : Operation {
  int Calculate (int operand1, int operand2) {
    return operand1 + operand2;
  }
}

[Extension]
[Plug("Operation")]
[Param("Symbol", "-")]
class Subtract : Operation {
  int Calculate (int operand1, int operand2) {
    return operand1 - operand2;
  }
}

[Extension]
[Plug("Operation")]
[Param("Symbol", "^")]
class Xor : Operation {
  int Calculate (int operand1, int operand2) {
    return operand1 ^ operand2;
  }
}
```

Figure 116: Contributors used in experimental evaluation

5.4.2 Seeded faults

We seeded seven faults in the Fibonacci testee. The following subsections describe these faults, explain their causes, and show what a correct implementation would look like.

5.4.2.1 Predictable order vs. unpredictable order fault (same contract)

Figure 117 shows the *StoreContributors* method of the testee, which is called when a contributor is plugged. The testee stores the first contributor as an add operation and the second contributor as a subtract operation, regardless of which operation the contributor actually provides. A correct implementation would retrieve the *Symbol* parameter value and assign the contributors accordingly.

```
class FibOperation : ... {
  Operation add, subtract;
  void StoreContributors(CompositionEventArgs args) {
    if (add == null) {
      add = (Operation) args.Plug.Extension.Object;
    } else if (subtract == null) {
      subtract = (Operation) args.Plug.Extension.Object;
    }
    // correct implementation:
    // switch ((String) args.Plug.Params["Symbol"].Value) {
    //   case "+":
    //     add = (Operation) args.Plug.Extension.Object;
    //     break;
    //   case "-":
    //     subtract = (Operation) args.Plug.Extension.Object;
    //     break;
    // }
  }
  ...
}
```

Figure 117: Predictable order vs. unpredictable order fault (same contract) in the testee

5.4.2.2 Duration fault

Figure 118 shows the *DisposeContributors* method of the testee, which is called when a contributor is unplugged. The testee does some cleanup work (not shown here), however it does not set the field for the removed contributor (add or subtract) to *null* as a correct implementation would.

```
class FibOperation : ... {
  Operation add, subtract;
  void DisposeContributor(CompositionEventArgs args) {
    Object contributor = args.Plug.Extension.Object;
    ... cleanup (cf. Figure 119) ...
    // correct implementation would include:
    // if (add == contributor) { add = null; }
    // else if (subtract == contributor) { subtract = null; }
  }
  int Evaluate(int operand) {
    ...
    switch (n) {
      case 0: case 1: return n;
      default: return add.Calculate(
          Evaluate(subtract.Calculate(n, 2)),
          Evaluate(subtract.Calculate(n, 1)));
    }
  }
  ...
}
```

Figure 118: Duration fault in the testee

5.4.2.3 Contributor identification fault

Figure 119 shows the *DisposeContributor* method of the testee, which is called when a contributor is unplugged. The testee casts the contributor to *IDisposable*, which is illegal because this is not included in the slot definition, i.e., a contributor is not required to implement this interface.

```
class FibOperation : ... {
  void DisposeContributor(CompositionEventArgs args) {
    Object contributor = args.Plug.Extension.Object;
    ((IDisposable) contributor).Dispose();
    // correct implementation would not dispose contributors,
    // because the Dispose method is not included in the slot
    // definition
  }
  ...
}
```

Figure 119: Contributor identification fault in the testee

5.4.2.4 Composition state mismatch

Figure 120 shows the constructor of the testee and the *StoreContributors* method, which is called when a contributor is plugged. The testee stores the first two contributors into fields (*add* and *subtract*) and ignores any further contributors contributors (i.e., it does neither store further contributors in fields nor intends to use them, although they would be plugged into the *Operation* slot). Furthermore, a correct implementation would use a composition behavior to make sure that only one add and one subtract contributor is plugged.

```
class FibOperation : ... {
  FibOperation(Extension self) {
    this.self = self;
    // a correct implementation would install a composition
    // behavior to prevent unnecessary contributors from
    // being plugged
  }
  Operation add, subtract;
  void StoreContributors(CompositionEventArgs args) {
    if (add == null) {
      add = (Operation) args.Plug.Extension.Object;
    } else if (subtract == null) {
      subtract = (Operation) args.Plug.Extension.Object;
    }
  }
  ...
}
```

Figure 120: Composition state mismatch in the testee

5.4.2.5 Contributor call in non-runtime thread

Figure 121 shows the *Evaluate* method of the testee. For numbers greater than five, the testee parallelizes the calculation using worker threads. When such a worker thread re-enters the *Evaluate* method, it accesses the operation contributors, which causes a Plux runtime error. A correct implementation would not call the contributors outside the Plux runtime thread.

```
class FibOperation : ... {
  Operation add, subtract;

  int Evaluate(int operand) {
    ...
    if (operand > 5) {
      int n1 = 0, n2 = 0;
      var n1Thread = new Thread(delegate() {
          n1 = Evaluate(subtract.Calculate(n, 1)); };
      var n2Thread = new Thread(delegate() {
          n2 = Evaluate(subtract.Calculate(n, 2)); };
      n1Thread.Start(); n2Thread.Start();
      n1Thread.Join();  n2Thread.Join();
      return add.Calculate(n1, n2);
    } else {
      switch (n) {
        case 0: case 1: return n;
        default: return add.Calculate(
            Evaluate(subtract.Calculate(n, 2)),
            Evaluate(subtract.Calculate(n, 1)));
      }
    }
  }
  ...
}
```

Figure 121: Contributor call in non-runtime thread in the testee

5.4.2.6 Single mandatory vs. multiple contributor cardinality fault

Figure 122 shows the *Evaluate* method of the testee. It accesses the contributors without prior null check, which can cause a null pointer exception depending on the composition state. A correct implementation would check the fields *add* and *subtract* for null references and would raise an *InvalidOperationException*. Please note, that according to the documentation of the slot definition *Function*, contributors are supposed to do so if required contributors are unavailable.

```
class FibOperation : ... {
  Operation add, subtract;
  void StoreContributors(CompositionEventArgs args) {
    if (add == null) {
      add = (Operation) args.Plug.Extension.Object;
    } else if (subtract == null) {
      subtract = (Operation) args.Plug.Extension.Object;
    }
  }
  int Evaluate(int operand) {
    // correct implementation:
    // if (add == null || subtract == null) {
    //   throw new InvalidOperationException(...);
    // }
    ...
    switch (n) {
      case 0: case 1: return n;
      default: return add.Calculate(
          Evaluate(subtract.Calculate(n, 2)),
          Evaluate(subtract.Calculate(n, 1)));
    }
  }
  ...
}
```

Figure 122: Single mandatory vs. multiple contributor cardinality fault in the testee

5.4.2.7 Use of not-plugged component

Figure 123 shows the *Evaluate* method of the testee. If *Evaluate* is called and the field *subtract* is *null* because the contributor was not plugged, the testee accesses the type store and creates a subtract contributor itself. When the testee calls a method on this self-created contributor, Plux raises a runtime error because the contributor is not plugged into the testee. A correct implementation would not create contributors itself but would raise an *InvalidOperationException* if required contributors were unavailable.

```
class FibOperation : ... {
  Composer composer;
  TypeStore typeStore;
  Operation add, subtract;
  FibOperation(Extension self) {
    Runtime runtime = self.Runtime;
    composer = runtime.Composer;
    typeStore = runtime.TypeStore
  }
  int Evaluate(int operand) {
    if (subtract == null) {
      foreach (PlugType pt in typeStore.GetPlugTypes("Operation")){
        if (((String) pt.Params["Symbol"].Value) == "-") {
          subtract = (Operation) composer
              .Create(pt.ExtensionType).Object;
          break;
        }
      }
    }
    ...
    switch (n) {
      case 0: case 1: return n;
      default: return add.Calculate(
          Evaluate(subtract.Calculate(n, 2)),
          Evaluate(subtract.Calculate(n, 1)));
    }
  }
  ...
}
```

Figure 123: Use of not-plugged component fault in the testee

5.4.3 Execution of the experiment

The experiment was conducted with two groups of students, with five students in each group. Group 1 tested with Actor, group 2 without Actor. The students had 90 minutes time to find as many faults as possible. We tracked which errors the students found and we conducted a survey where the students self-assessed how many errors they found and how confident they were that all faults were found.

Group 1 found 4.8 errors on average, whereas group 2 found only 2.6 errors. As a group, group 1 found all seeded faults, whereas group 2 missed 3 out of 7 faults. The results indicate four kinds of faults: faults which can easily be found, regardless of Actor support (faults A and C); faults which are easier to find with Actor than without it (faults F and G); faults which are hard to find, but were found equally often regardless of Actor support (fault D); and faults which are generally hard to find and were only found with Actor (faults B and E). The self-assessed confidence in group 1 was higher than in group 2 (2.2 vs. 1.6). Figure 124 shows

the results in detail. Please note, that some students reported more found faults than they actually found, because they counted the same fault multiple times.

	Student	Fault found							Actual number of faults found	Self-assessment	
		A	B	C	D	E	F	G		Number of faults found*	Confidence** [1-4] more is better
Group 1 using Actor	1	X	-	X	-	-	X	-	3	4	1
	2	X	-	X	X	-	X	X	5	5	3
	3	X	-	X	-	-	X	X	4	5	2
	4	X	-	X	-	X	X	X	5	6	2
	5	X	X	X	X	X	X	X	7	8	3
Total group 1	1-5	5	1	5	2	2	5	4	4.8	5.6	2.2
Group 2 without Actor	6	X	-	-	X	-	-	-	2	1	1
	7	X	-	X	-	-	X	-	3	4	2
	8	-	-	X	X	-	-	-	2	2	1
	9	X	-	X	X	-	-	-	3	4	2
	10	X	-	X	-	-	X	-	3	7	2
Total group 2	6-10	4	0	4	3	0	2	0	2.6	3.6	1.6

Faults

A Specific reliable order vs. unreliable order fault (same contract)
B Duration fault
C Contributor identification fault
D Composition state mismatch
E Contributor call in non-runtime thread
F Single mandatory vs. multiple contributor cardinality fault
G Use of not-plugged component

Self-assessment

* How many faults have you found?

** How confident are you that you found all faults?

1 not at all (missed most)
2 a little (missed some)
3 quite (found most)
4 most (found all)

Figure 124: Results of the experimental evaluation

Chapter 6: Locating the cause of composition errors

The composability test method finds errors in components by executing a large number of test cases. In order to fix a faulty component, developers must debug it and find out which fault causes the error. Typically the same error is found by multiple test cases. One approach to find out the cause of an error is to analyze the commonalities of the test cases that revealed the error as well as the differences to other test cases.

In this chapter, we present a debugging method based on the above approach for the run-time injection with tracking composition mechanism. The debugging method records the composition of a program, analyzes composition operation sequences and composition states, and hints at possible causes of a composition error. We show the application of this debugging method for Plux components and present a debugging tool for Plux.

6.1 The composition debugging method Doc

The composition debugging method (*Doc*) analyzes the composition state and the composition operations that produced that composition state in order to find out the cause of a composition error. Doc is a post-mortem debugging method, i.e., it analyzes the composition that was recorded during program execution.

The Doc method can be applied in two scenarios: it can analyze the test results produced by Act (*test mode*), or it can analyze a program execution where the user controls the program (*user mode*). In test mode, Doc support developers in finding the cause of errors revealed during testing; in user mode, Doc helps finding the cause of errors reported by users.

6.1.1 Recording composition operations

During the execution of a program, Doc records all composition operations in a composition trace. In test mode, Act executes test cases and Doc records a composition trace for each test case. In user mode, the user executes a program and Doc records the composition trace while the program is running. Doc records all composition operations into the trace. Figure 125 shows an example of a compo-

sition trace recorded in a library application (for shortness only the *Plug* and *Unplug* operations are shown).

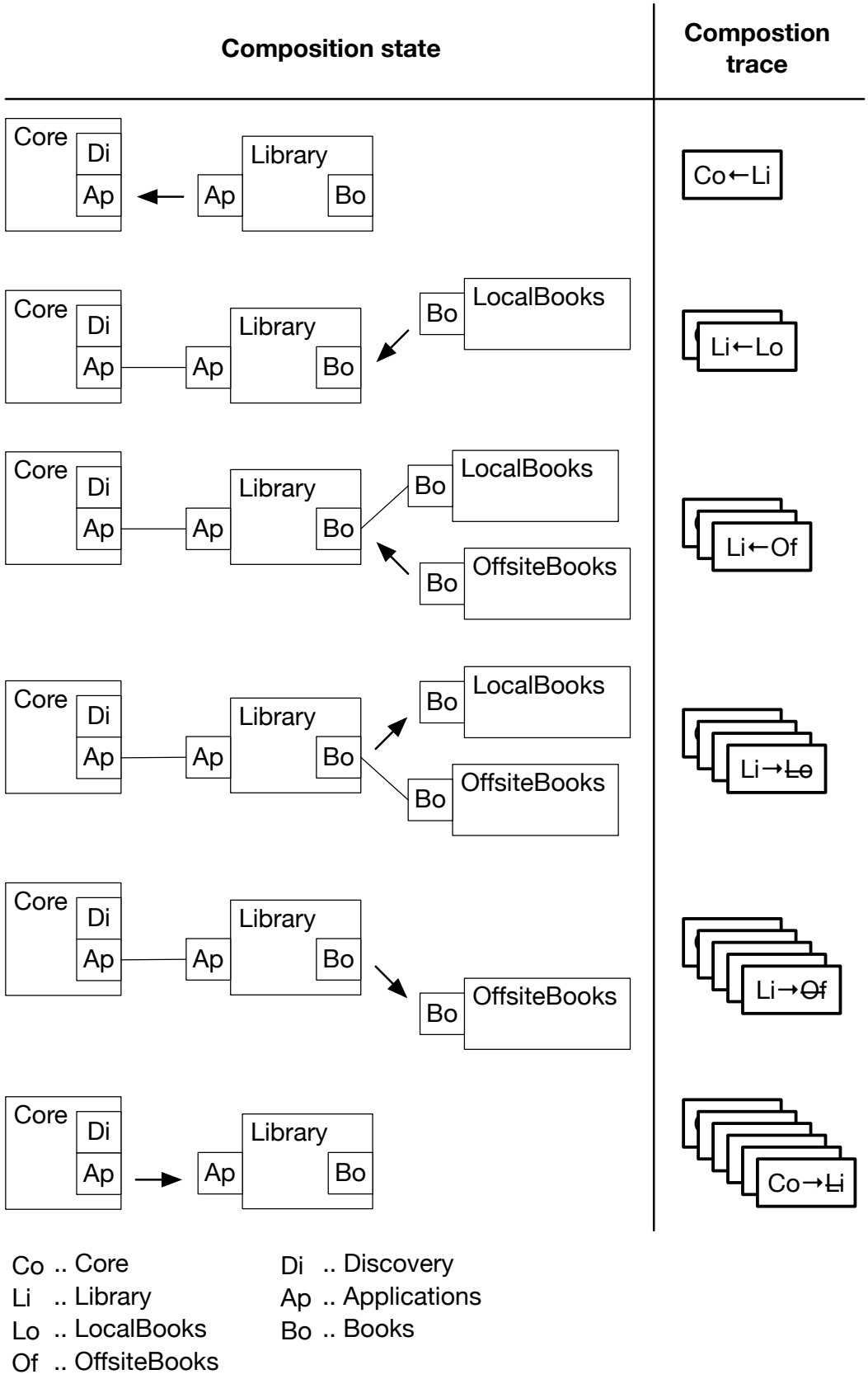


Figure 125: Recording composition operations

6.1.2 Filtering composition operations

In test mode all recorded composition operations are relevant for debugging, but in user mode the program usually consists of a large number of components, many of which are irrelevant for the error that occurred. To focus on the relevant composition operations, Doc provides the following filters:

Extension type filter	..	Selects the composition operations for all instances of the specified extension type.
Extension instance filter	..	Selects the composition operation for the specified extension instance.
Composition state filter	..	Selects the composition operations for the specified host instance and all its (direct or indirect) contributors.
Composite filter	..	Combines multiple filters, either by union (any filter matches) or by intersection (all filters match).

Figure 126 shows an example for composition trace filtering. The unfiltered composition trace (a) was recorded in user mode and shows the composition of the library with the local books and offsite books contributors. The first composition of the library host (time 3 with *Li*←*Lo* until 10 with *Li*→*Of*) does not reveal an error, whereas the second (time 14 with *Li*←*Of* until 21 with *Li*→*Of*) shows an error. As the *Core* is irrelevant for this error, it is filtered out using the composition state filter with the *Library* as root (b).

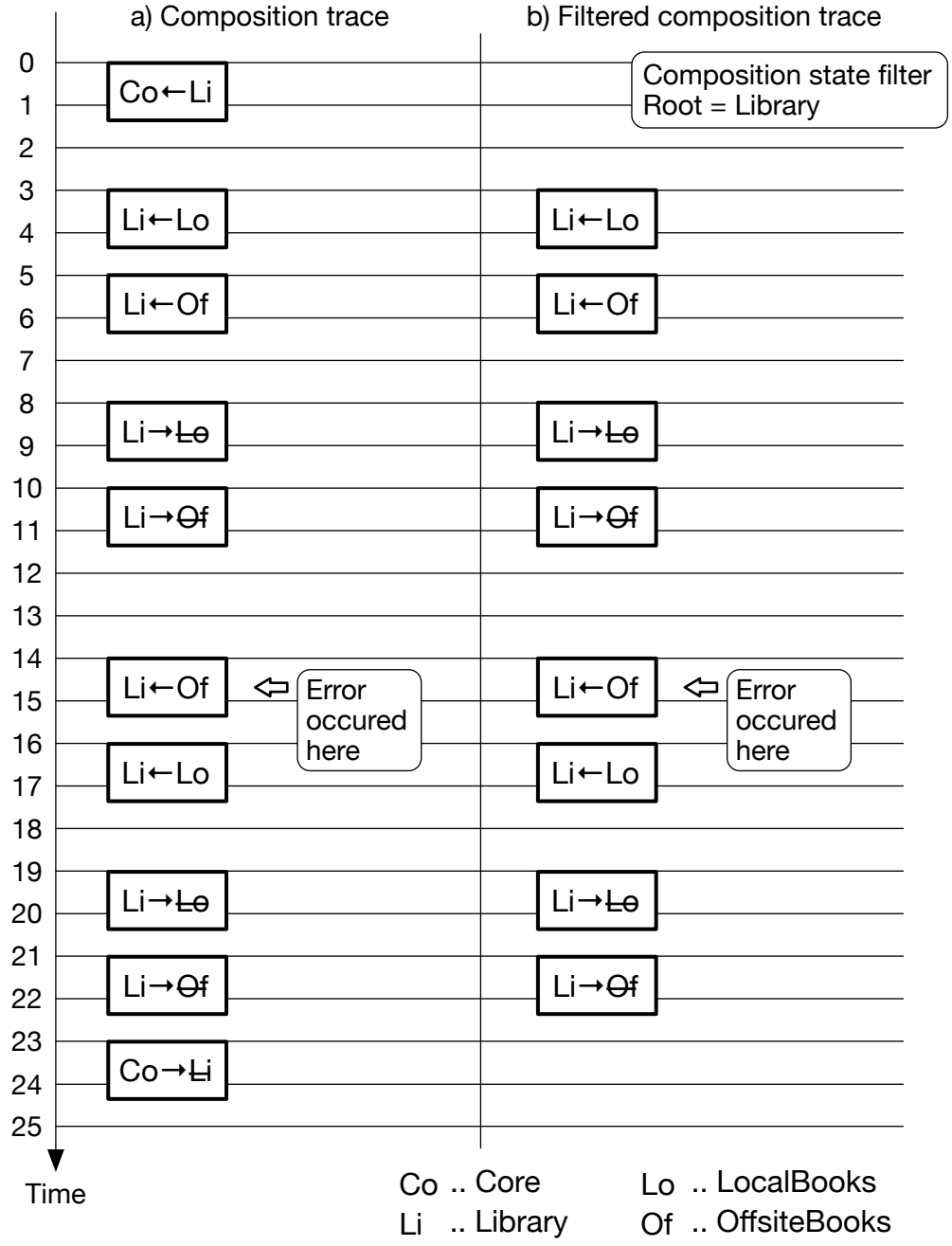


Figure 126: Filtering the composition operations

6.1.3 Splitting composition traces

Doc can divide a composition trace into parts that contain related composition operations. Each part typically contains a different composition sequence for the faulty component, which can be compared with other parts in order to analyze the fault. This is useful in user mode, where a composition trace originally contains a whole program execution and needs to be split into parts before composition sequences can be compared. In order to split a composition trace, Doc searches for clusters of composition operations that are chronologically related, using the *k-means clustering algorithm* [Lloyd, 1982]. This is reasonable, because during pro-

gram execution a sequence of composition operations that belong together is typically followed by a state where the composition is idle, because the program waits for user input or executes operations. The fact that the idle phase is typically longer than the composition phase allows building clusters of related composition operations.

Figure 127 shows how the composition trace for the library is split into parts (cf. Figure 126 for the trace before the split). Doc found three phases where composition was idle (6-8, 11-14, 17-19) and thus identified four composition operation clusters. So the composition is split into four parts. Parts 1 and 3 compose the library, parts 2 and 4 decompose it.

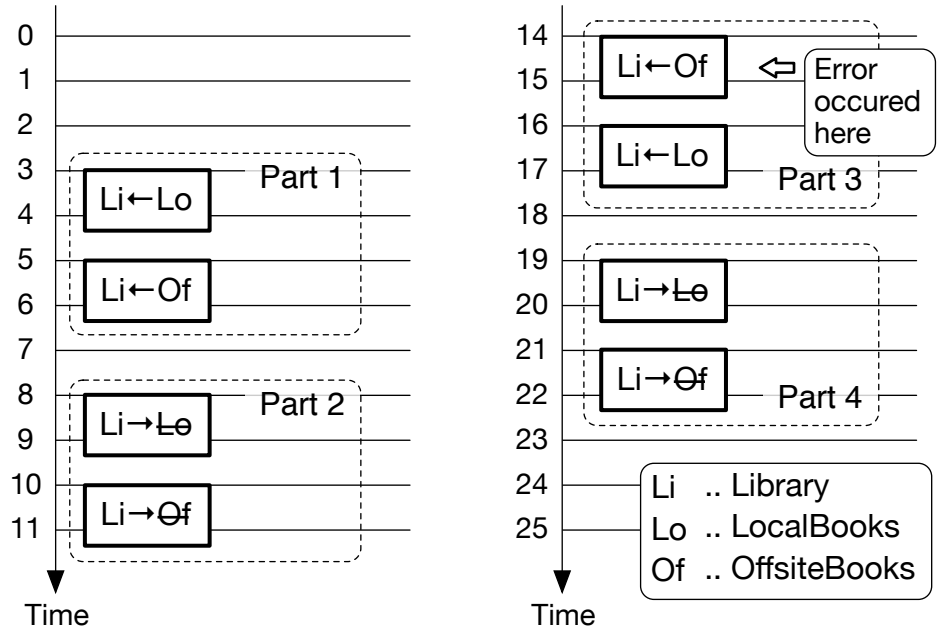


Figure 127: Splitting the composition trace

6.1.4 Comparing composition traces

Doc can compare composition traces (or parts of a trace) in order to find the composition state in which a faulty component shows an error. Doc also visualizes the differences between two composition traces. To determine these differences Doc uses the *Needleman-Wunsch* [Needleman and Wunsch, 1970] algorithm, which aligns the composition operations of two traces by adding gaps or by setting mismatches in the alignment. It compares the total costs for all possible combinations of gaps and mismatches and determines an optimal combination, i.e., an alignment with the lowest possible costs. The algorithm is configured with the costs for gaps, mismatches, and matches. Depending on this configuration, the algorithm finds a more or less compact alignment. A compact alignment contains more mismatches, whereas a sparse alignment contains more gaps.

Figure 128 shows how two composition trace parts for the library are aligned. In this example, the user compared part 3 (because it showed the error) with part 1 (because it also composes the library). The goal is to find the swapped composition operations as shown in (a), because this is likely related to the cause of the error. In the first alignment, where Doc detects two mismatches (b), the user can see that the traces are different, but it is difficult to see the swap. In the second alignment, where Doc inserted two gaps, the swap is easier to see, because one can see that the composition operation *Li←Of* is equal in both traces and that *Li←Lo* occurs before it in part 1 and after it in part 2. As these swapped composition operations are likely the cause for the error, the user must inspect the implementation that handles the composition events for the books contributors in the library.

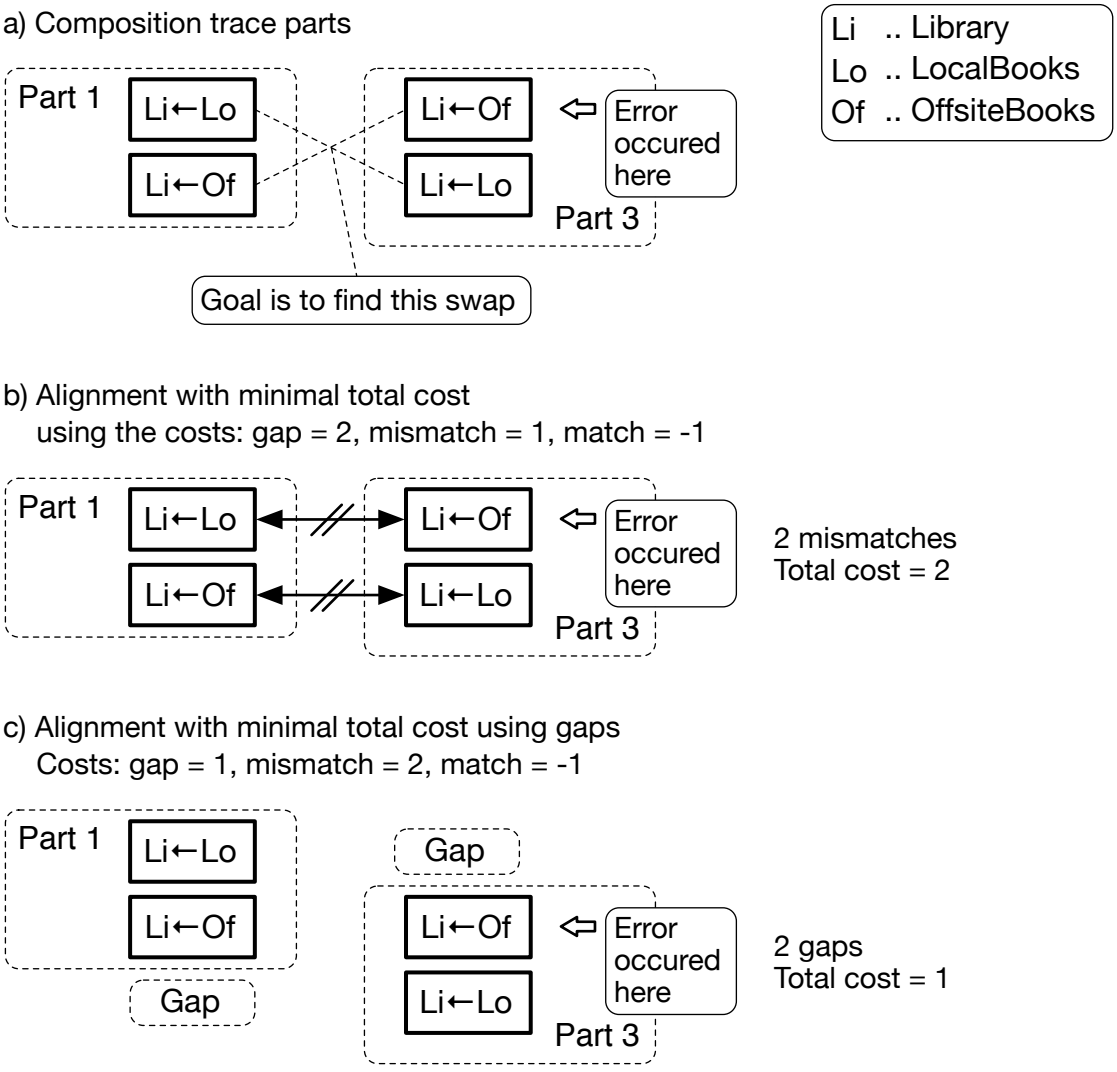


Figure 128: Comparing composition traces

6.1.5 Reasoning about the error causes

Doc can generate hints for possible error causes using reasoning. It generates language automata from reference traces (without errors) as well as from error traces, and creates hints by comparing these automata.

Doc generates the language automata as follows: it starts with an empty composition state; it uses the first composition operation from the trace as transition into the next composition state; this procedure is repeated for all composition operations in the trace. If the composition trace contains composition sequences that lead to equal composition states (i.e., states with the same components), the same composition state is used in the automaton, i.e., these composition sequences transition to the same composition state.

The hints are created as follows: Doc starts at the final composition state of the error automaton and finds all composition operations that transition into this state; it looks up the corresponding composition operations in the reference automaton and compares the source composition states in both automata; finally, it prints the differences between the source composition states as a human-readable hint. If the composition operation causing the error appears repeatedly in the composition trace, and the corresponding source composition states are different, Doc generates multiple hints.

In order to rank the hints, a confidence is calculated. This confidence corresponds to the fraction of composition states (please note, only composition states on which the according composition operation is executed) that contain the hinted extension (i.e., the extension that is likely to trigger the error in the composition operation) within all composition states, if that extension is missing in the error case, i.e., confidence = states with hinted extension / (states with hinted extension + states without hinted extension). Otherwise, if the error case contains the hinted extension, the confidence is calculated the other way round, i.e., with the composition states that miss the hinted extension in the nominator.

Figure 129 shows the reasoning of the error cause for our library example. Part 1 is the reference composition trace and part 3 is the error composition trace (a). Doc generates a language automaton for both traces (b). Thereby it ignores any composition operation which comes after the error when generating the error automaton (shown with dashed lines). The reasoning starts at the final composition state in the error automaton (1); it follows the composition operation *Li←Of* back to the empty composition state (2); it looks up *Li←Of* in the reference automaton and follows it back to the *Lo* composition state (3); it compares the empty and the *Lo* composition states and creates the hint "*Li←Of* depends on

Lo" which means that the likely cause for this error is that the handler for the *Plugged* event in the library relies on the local library to be already plugged when the offsite library is plugged.

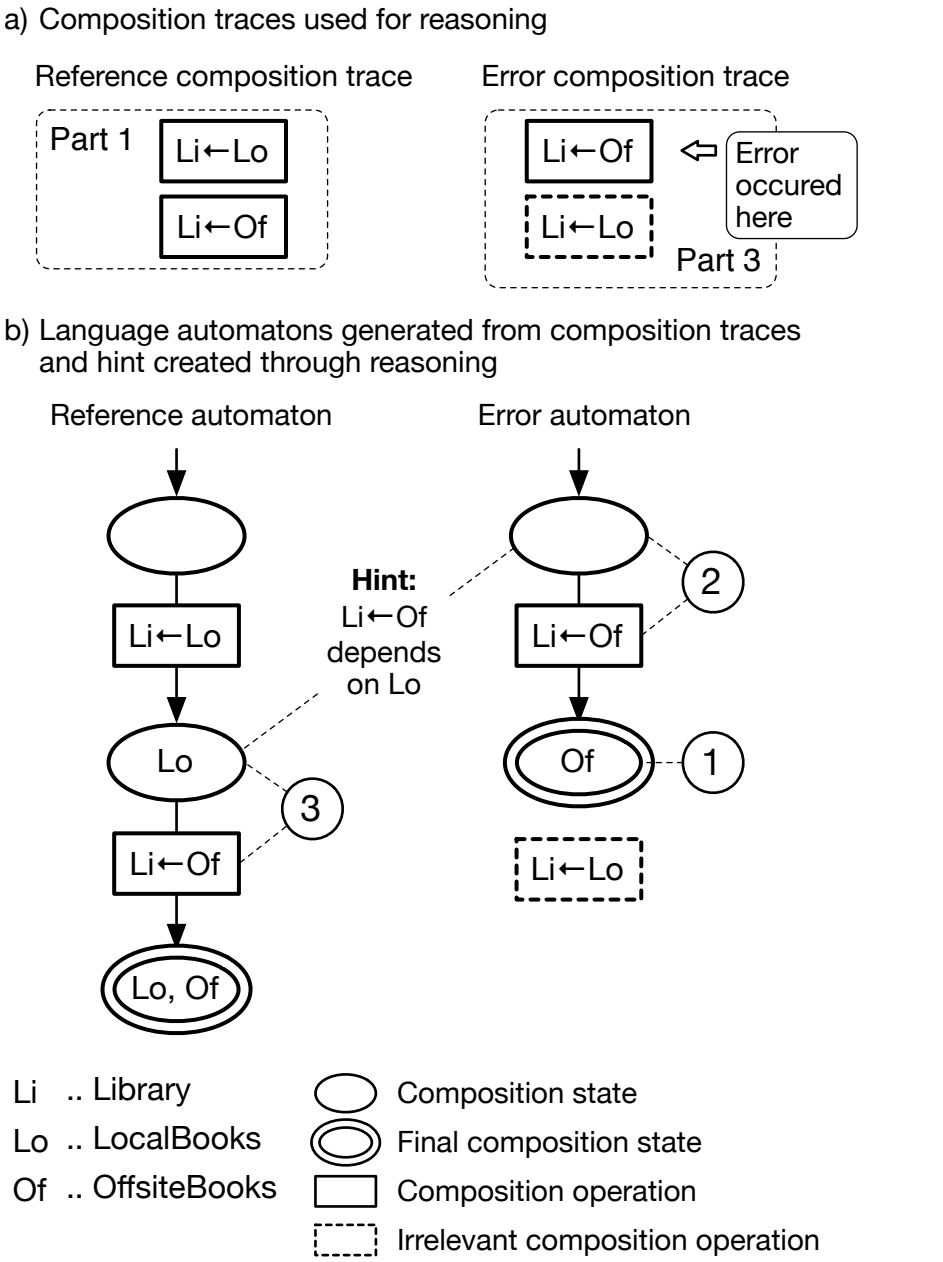


Figure 129: Reasoning the error cause

6.1.6 Replaying composition traces

Doc can replay composition traces in such a way that it visualizes the composition state of the program for every point in the composition trace. This is useful to see differences between composition traces that show an error and such that do not. Doc allows the user to replay a composition trace step by step and in both directions, i.e., one can step forwards and backwards. Each composition operation updates a virtual composition state (i.e., the program is not executed) which is visualized as a graph.

Figures 130 and 131 show the replaying of the library. In this example, the error composition trace (cf. Figure 130) and the reference composition trace (cf. Figure 131) are replayed. By comparing the visualization of both composition states (reference and error) one can see that the error occurs when *OffsiteBooks* is plugged before *LocalBooks* is plugged.

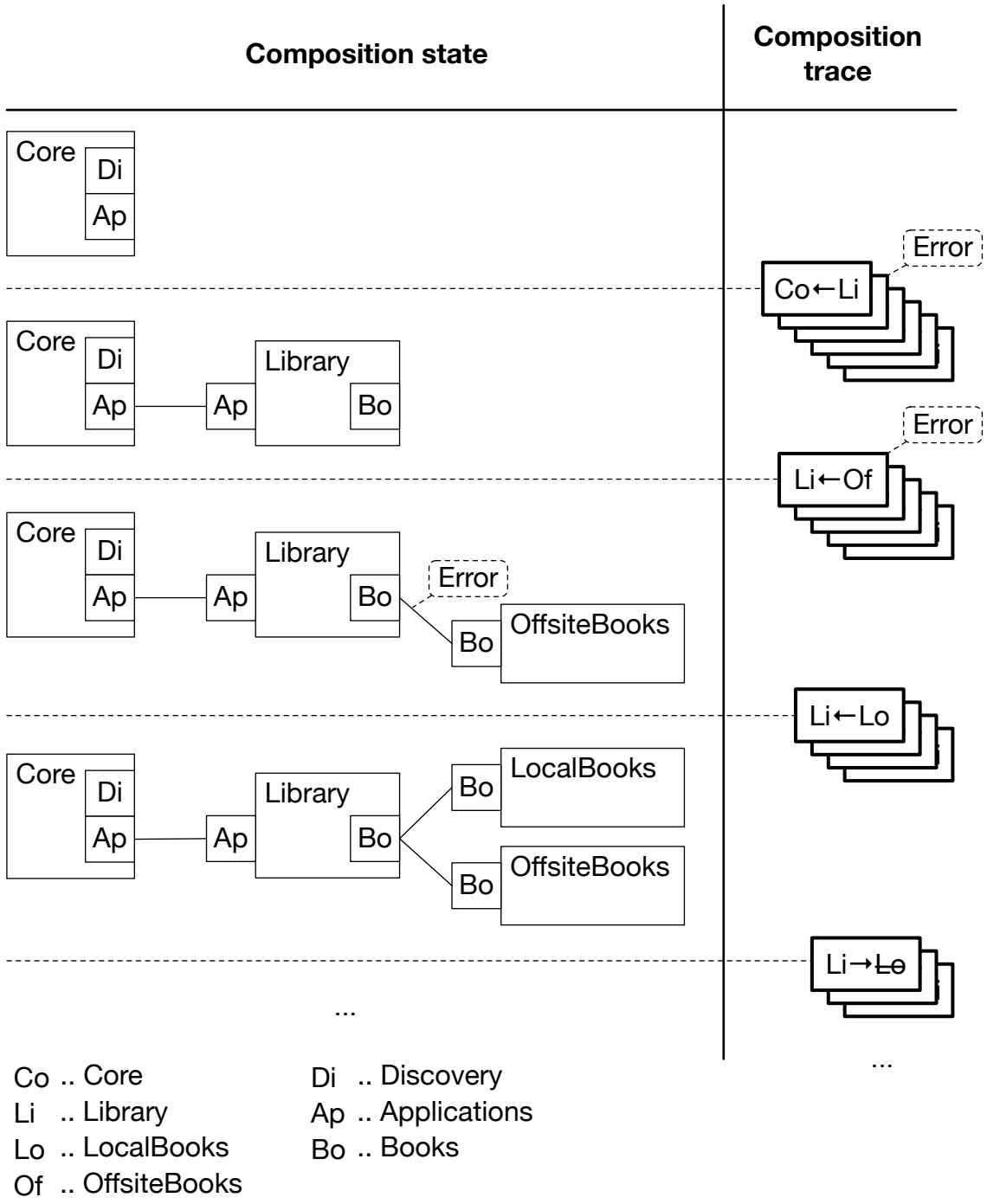


Figure 130: Replaying the error composition trace

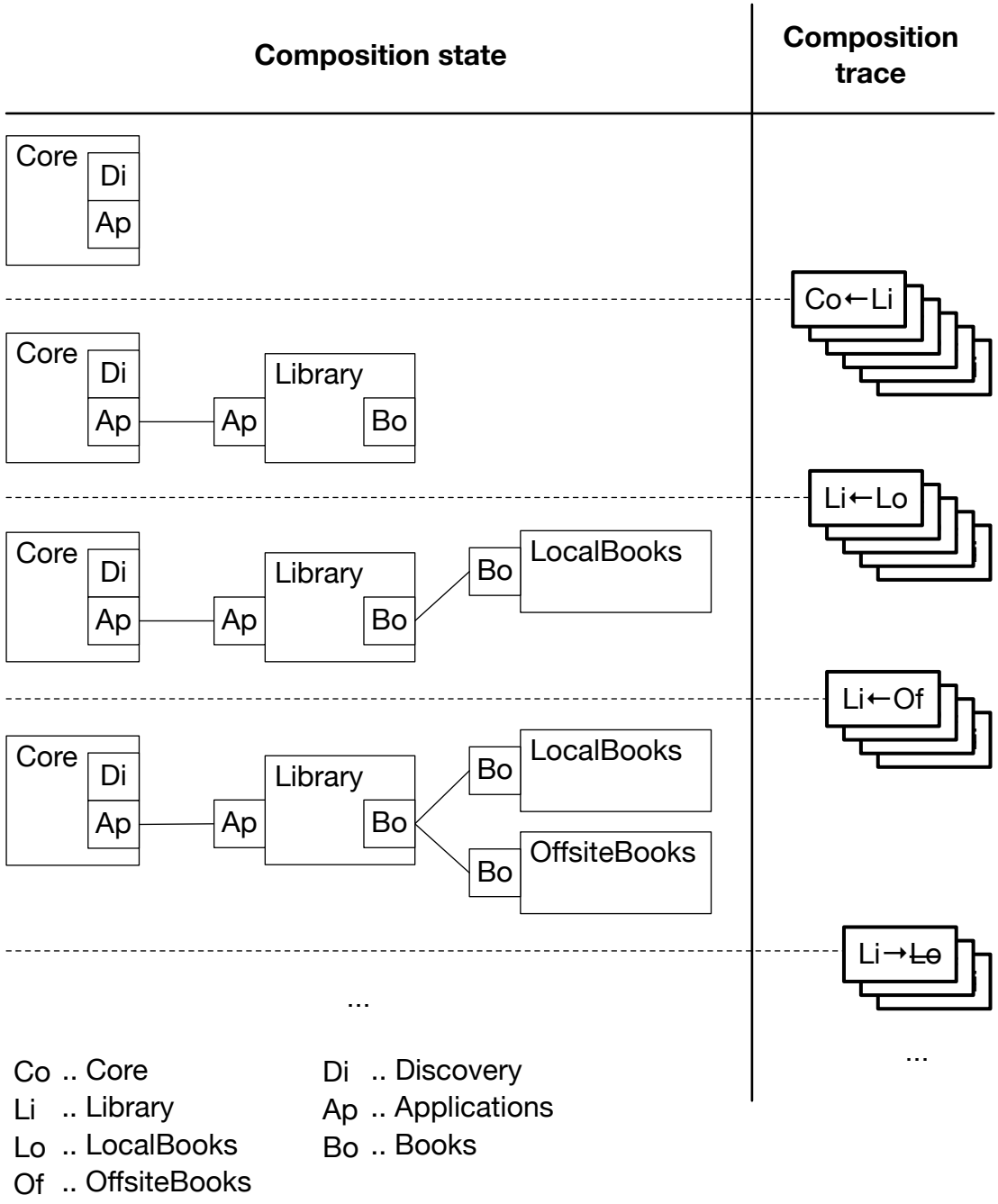
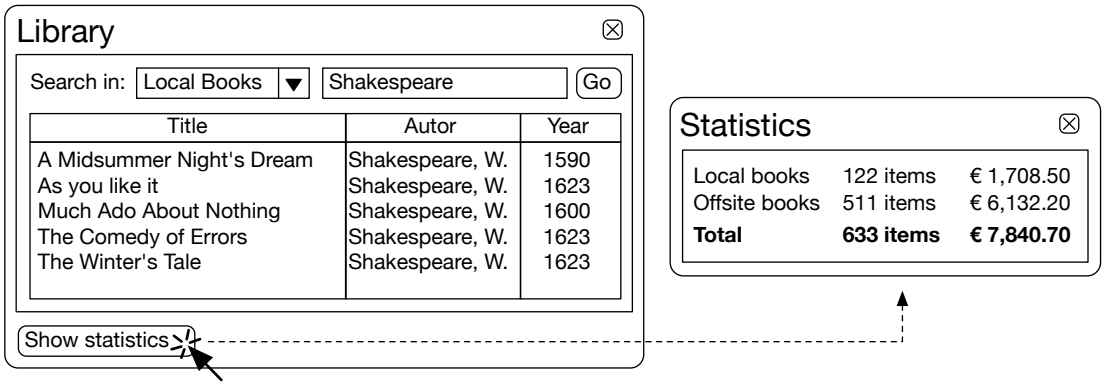


Figure 131: Replaying the reference composition trace

6.2 Debugging Plux programs

This section shows how to apply the composition debugging method Doc in order to find the cause of errors in the Plux library example. Figure 132 shows the user interface of the library application with a flawless calculation of the statistics (a) and a null pointer error (b).

a) Library statistics use without error



b) Library statistics use with error

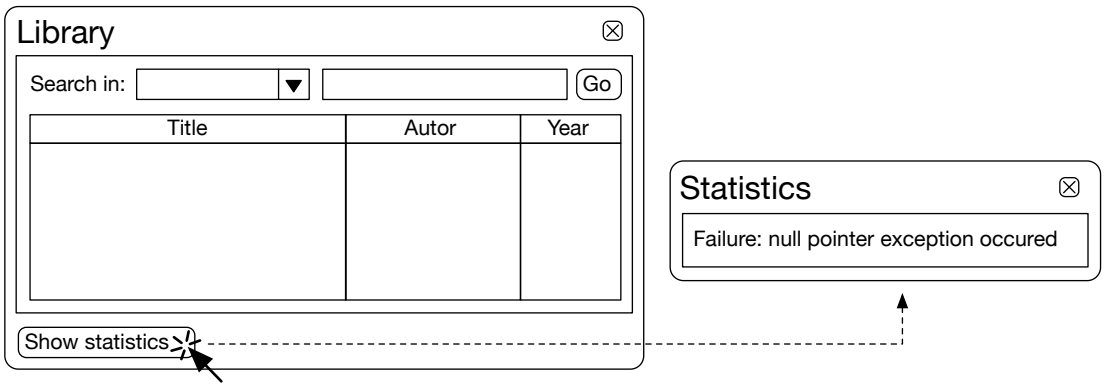


Figure 132: User interface of the library application with statistics

Doc recorded the composition trace (of a run showing an error) in Figure 133 (a). We filtered the composition trace using a composition state filter with the *Library* extension as the root (b) and split the composition trace between the uses of the library at time 222-450 (c). Part 1 is the reference composition trace and part 2 is the error composition trace.

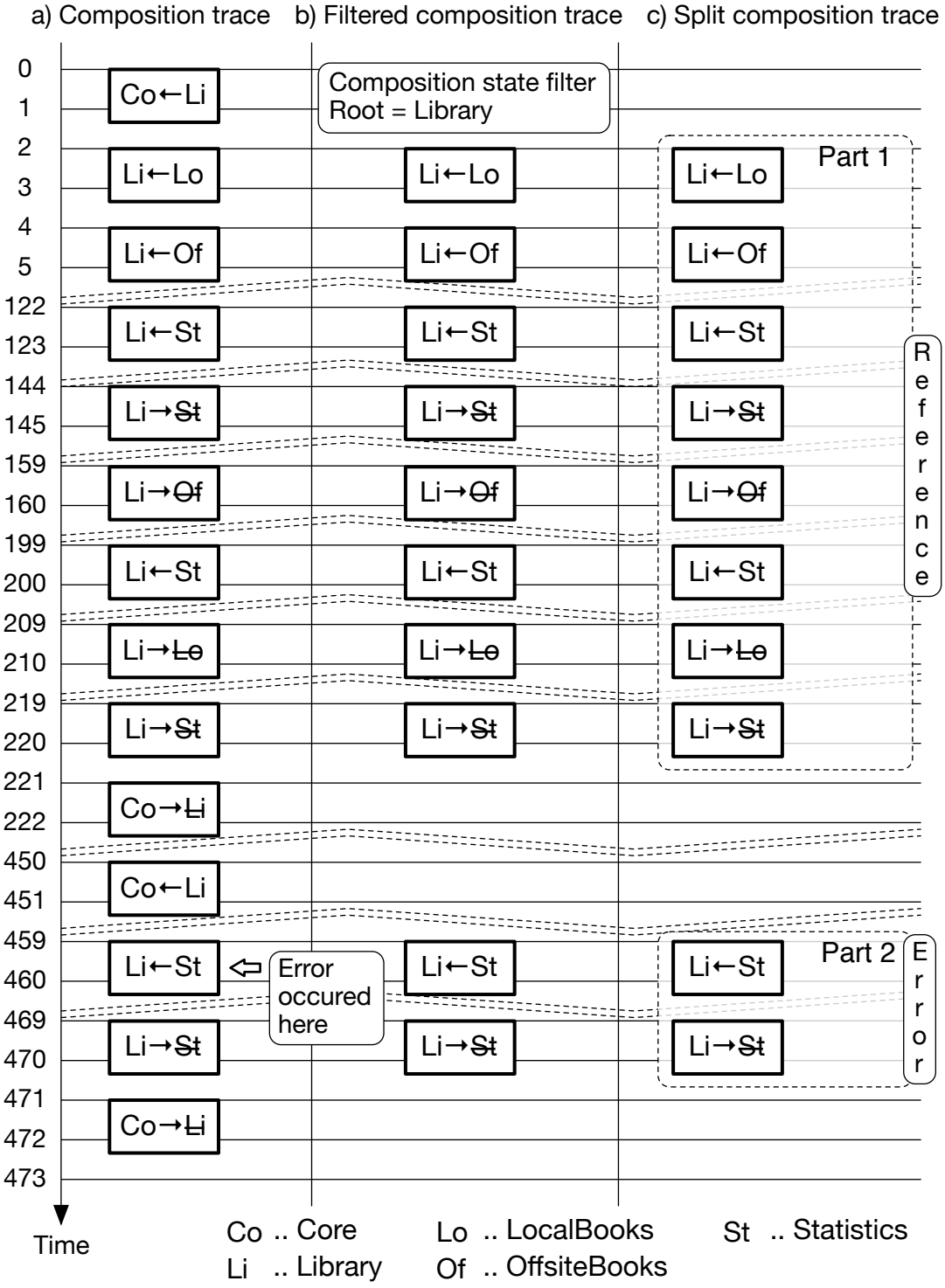


Figure 133: Filtering and splitting the composition trace of the library application

We compared the composition trace parts of the library application (cf. Figure 134). Doc aligned the composition trace parts by inserting gaps. We can already see that in the reference trace (a) the book store contributors *LocalBooks* and *OffsiteBooks* were plugged before the statistics tool, whereas in the error trace (b) the book store contributors were not plugged when the statistics tool was added.

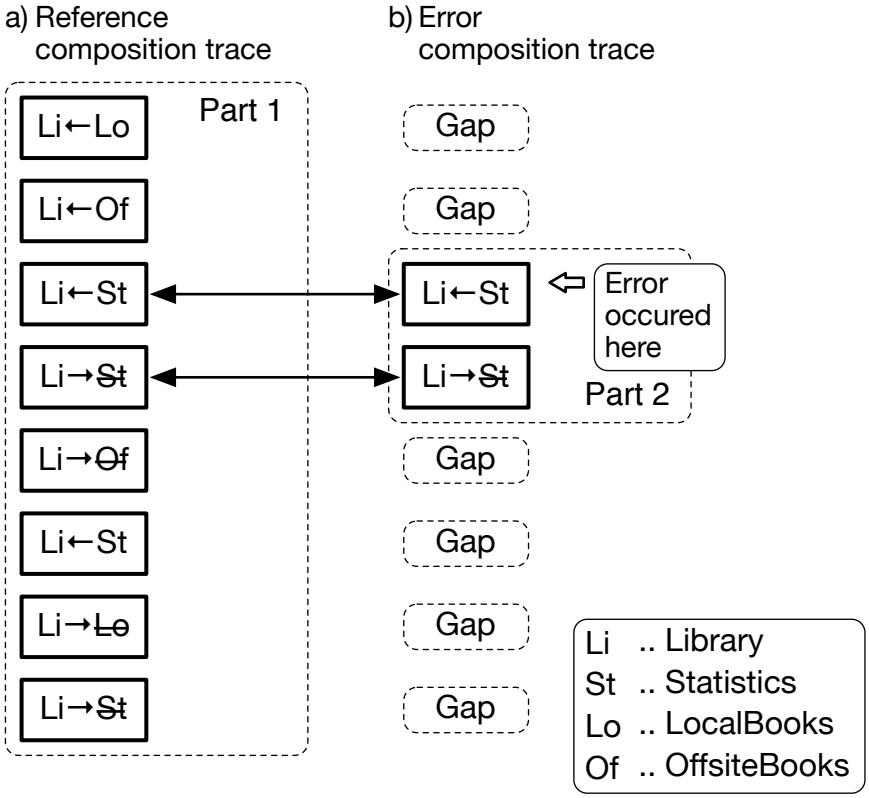


Figure 134: Comparing the composition trace parts of the library application

Doc generates the language automata (cf. Figure 135) for the reference (a) and the error (b) composition trace and uses them to infer the hints (c): the composition operation *Li←St* depends on the *LocalBooks* contributor with a confidence of 1 and on the *OffsiteBooks* contributor with a confidence of 0.5. These two hints indicate that the statistics tool requires a book store contributor plugged into the library. The reason for the different confidence is that the reference trace contains more relevant composition states that include the *LocalBooks* contributor than the *OffsiteBooks* contributor.

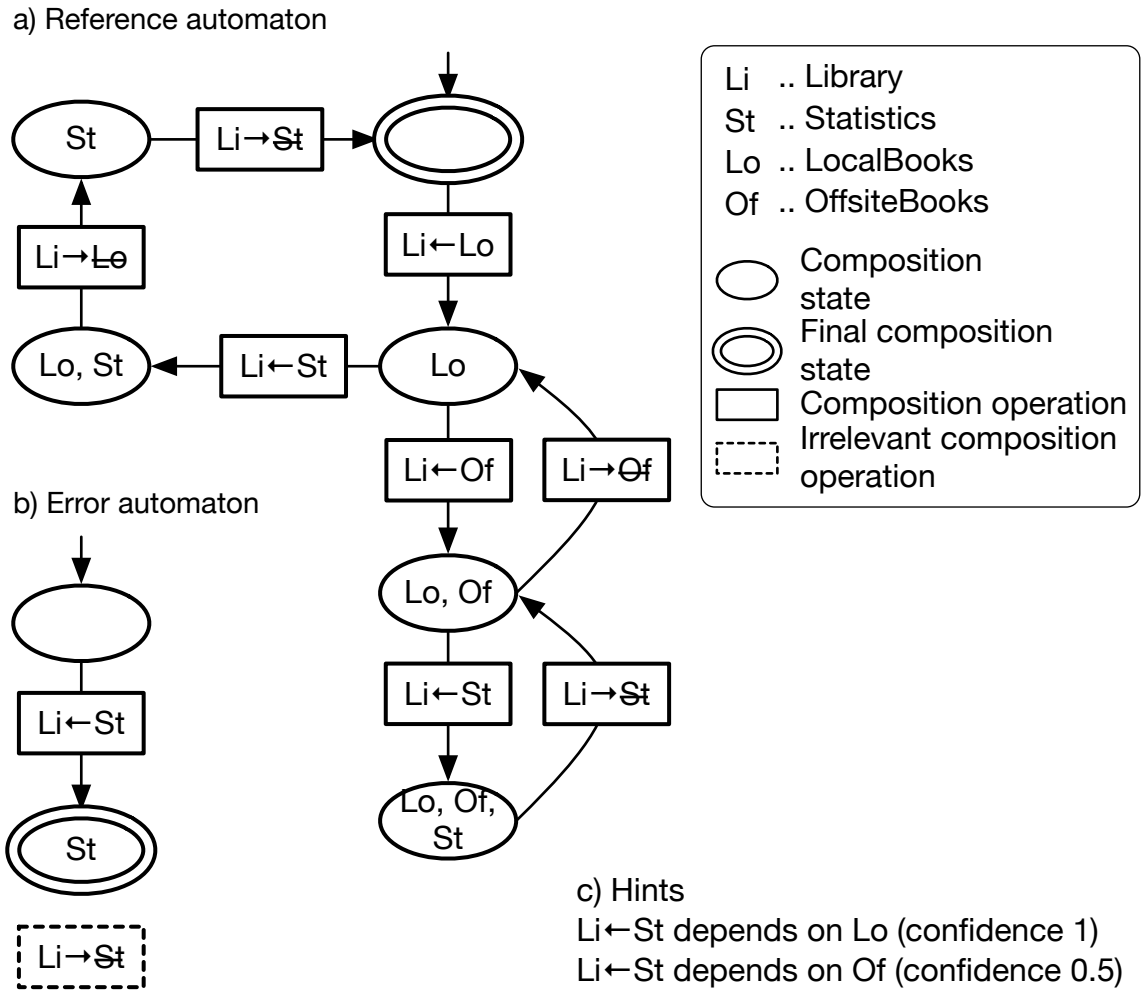


Figure 135: Reasoning about the error cause in the library application

In order to verify the hints we replayed the composition trace parts. Figure 136 shows the relevant sections of the replay where the statistics contributor is plugged into the library. In the replay of the error trace the *Books* slot is empty, whereas in the replay of the reference trace the *Books* slot is filled. This confirms the hints.

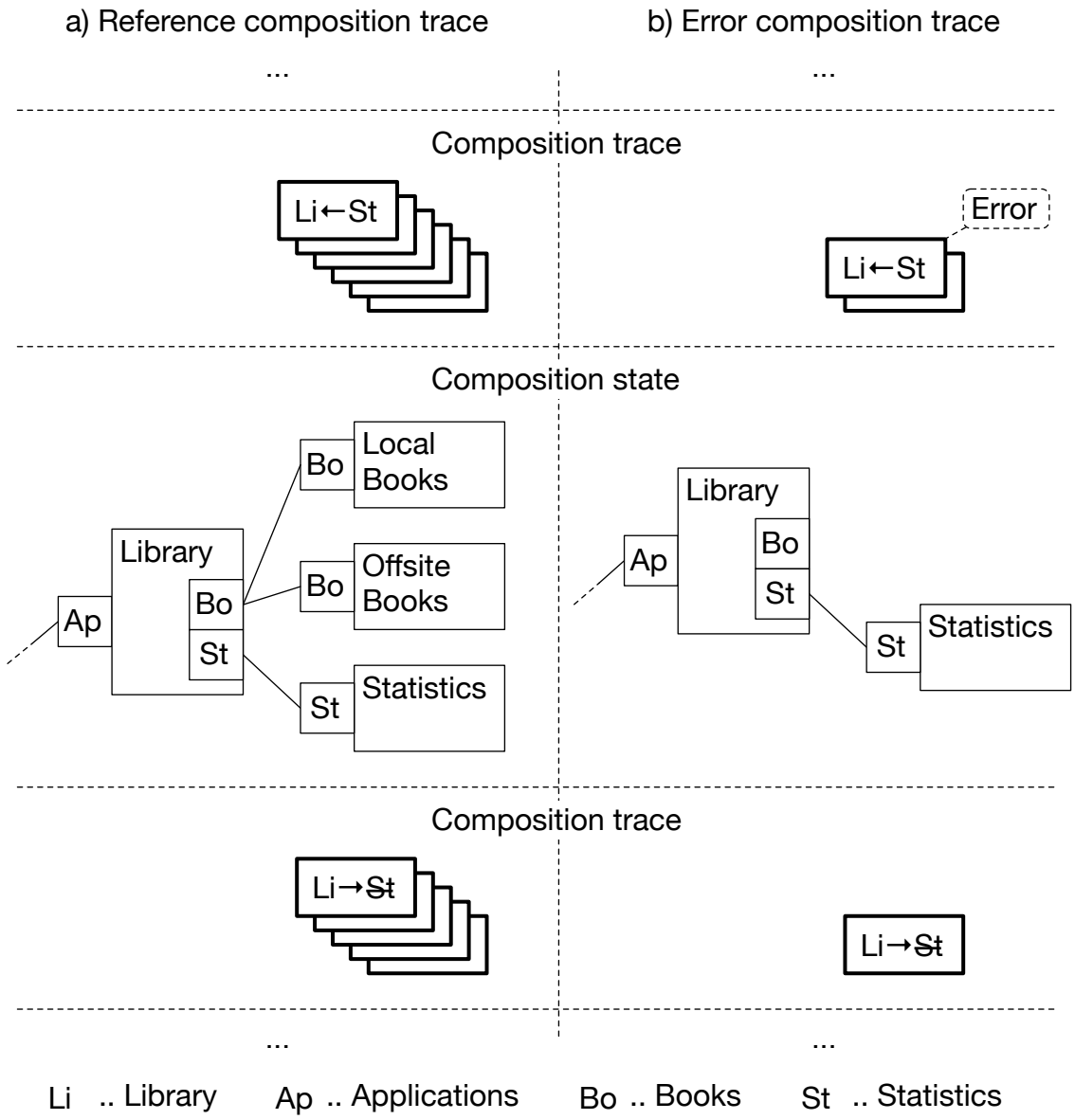


Figure 136: Replaying the composition trace parts of the library application

Let us finally check the implementation of the library host for the actual error cause (cf. Figure 137). The *Plugged* event of the *Statistics* slot calls the *SetStatistics* method. There the *bookStores* field is accessed without a prior null reference check, which causes the null pointer error in the composition state shown by the error composition trace. A correct implementation of the library host would check for the null reference.

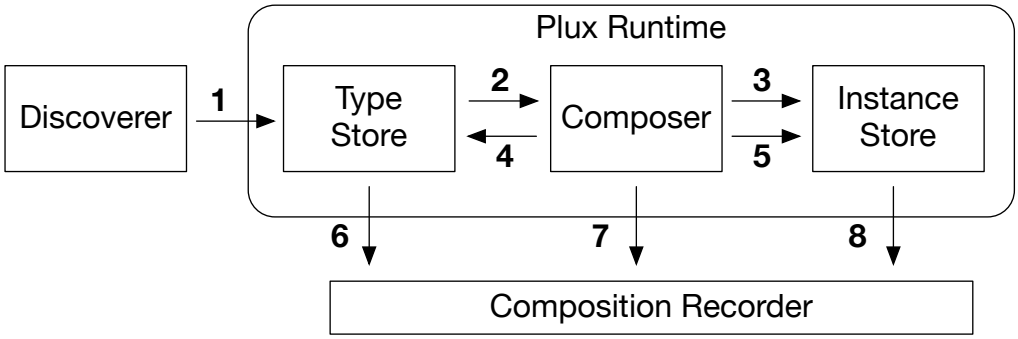
```
[Extension]
[Plug("Application")]
[Slot("Books", OnPlugged = "AddBookStore",
      OnUnpluging = "RemoveBookStore")]
[Slot("Statistics", OnPlugged = "SetStatistics")]
class Library : IApplication {
  List<Books> bookStores;
  Statistics statistics;
  void AddBookStore(CompositionEventArgs args) {
    if (bookStores == null) {
      bookStores = new List<Books>();
    }
    bookStores.Add((Books) args.Plug.Extension.Object);
  }
  void RemoveBookStore(CompositionEventArgs args) {
    bookStores.Remove((Books) args.Plug.Extension.Object);
    if (bookStores.Count == 0) {
      bookStores = null;
    }
  }
  void SetStatistics(CompositionEventArgs args) {
    statistics = (Statistics) args.Plug.Extension.Object;
    foreach (Books books : bookStores) {
      foreach (Book b : books) {
        statistics.Add(books.Id, b);
      }
    }
    // ...
  }
}
```

Figure 137: Implementation of the library application

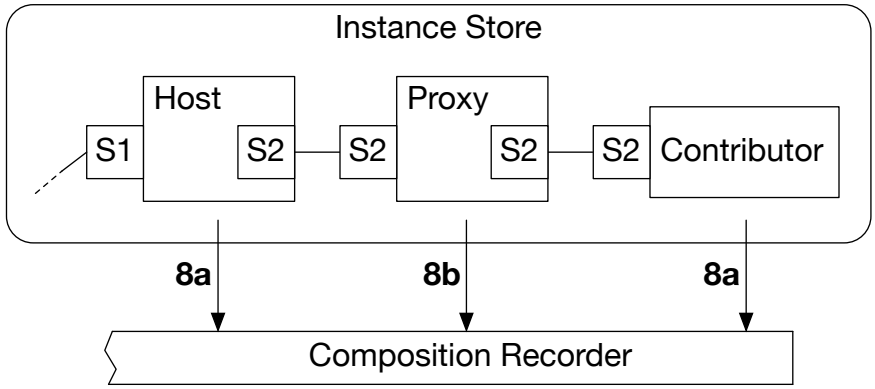
6.3 Composition debugging tool Doctor

The composition debugging tool *Doctor* implements the composition debugging method Doc from Section 6.1. This section gives only a short description of Doctor, a detailed description with implementation specifics can be found in [Lengauer, 2012]. Doctor records the composition operations from the Plux composition infrastructure. Figure 138a shows how the Doctor operation recorder is integrated into the Plux runtime to record the composition operations: (6) it records the discovery operations from the type store; (7) it records the composition operations from the composer; (8) for diagnostic purposes, it records the log messages from extensions that use the Plux log mechanism, e.g., when an extension fails to open a file (8a), and the method calls between extensions (8b). The recording happens while Plux executes the program (1 to 5 cf. Section 4.5). Figure 138b shows the integration of the composition recorder in the instance store in detail. Doctor installs a proxy between every host and contributor connection. This proxy wraps the methods of the contributor and records the method calls (including parameter values) from the host to the contributor.

a) Plux composition infrastructure and Doctor composition recorder



b) Instance store in detail



1 Adds and removes contracts and plugins (discovery)
2 Notifies on changes
3 Queries for matching slots
4 Queries for matching plugs
5 Stores instance metadata and relationships
6 Records discovery operations
7 Records composition operations
8 Records log (a) and call (b) operations

Figure 138: Composition debugging tool architecture

The command *Doctor Library.trace* invokes Doctor with the recorded composition trace of the library application. Figures 139 and 140 show the output of Doctor. Figure 139 shows on the left-hand side the comparison of the reference trace with the error trace and on the right-hand side the hints for the error cause created by reasoning. Figure 140 (left) shows the replay output of Doctor, i.e., the visualization of the virtual composition state (top) and the progress in the replayed error trace (bottom); on the right-hand side, it shows the detected composition standard violations. Doctor detected that the library extension calls the book store contributors in a non-runtime thread. Please note, this thesis uses simplified drawn user interfaces of Doctor, however, real screenshots are given in the Appendix, see Figures 143-146 on page 155ff.

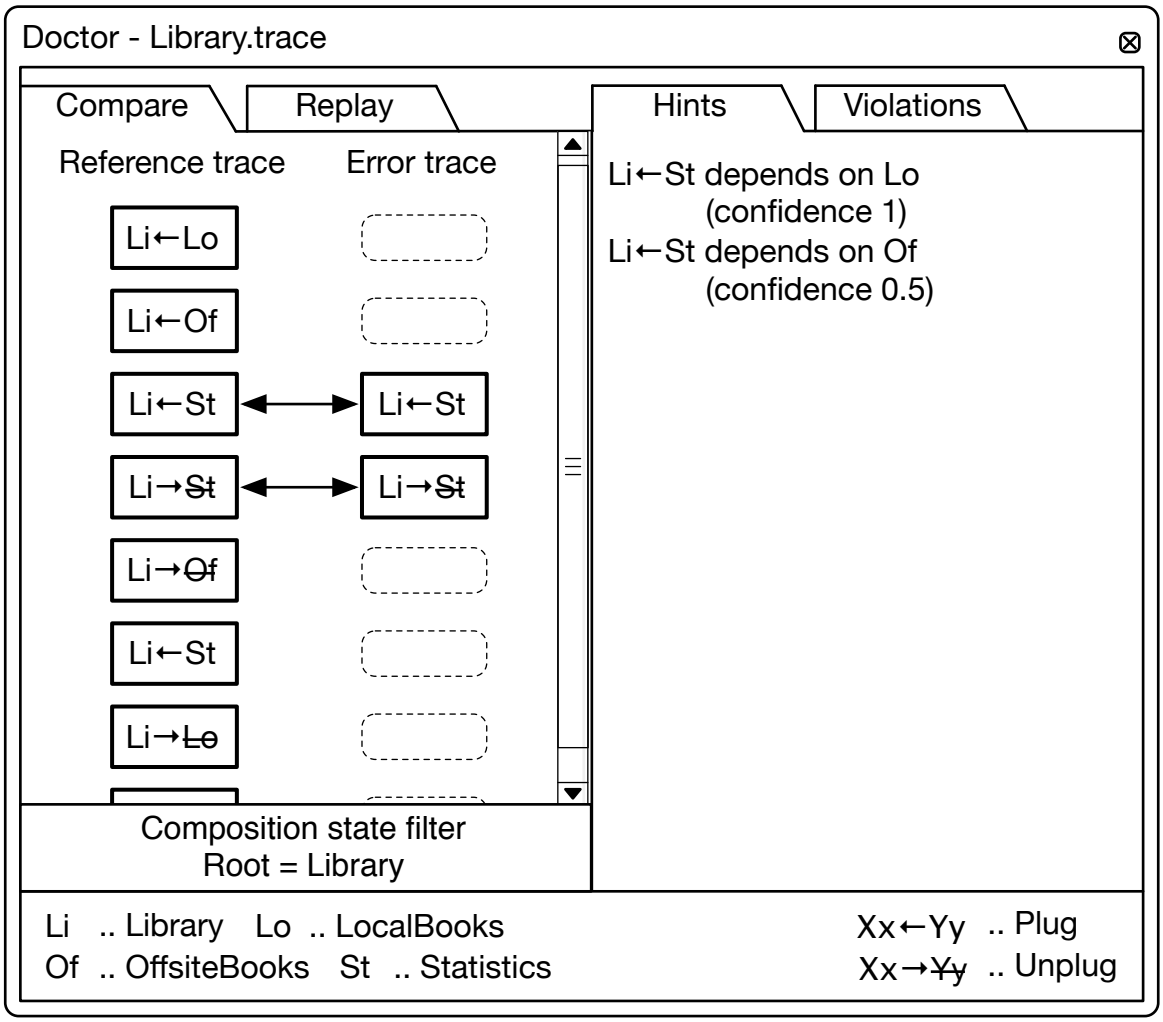


Figure 139: Composition trace comparison and hints in the composition debugging tool

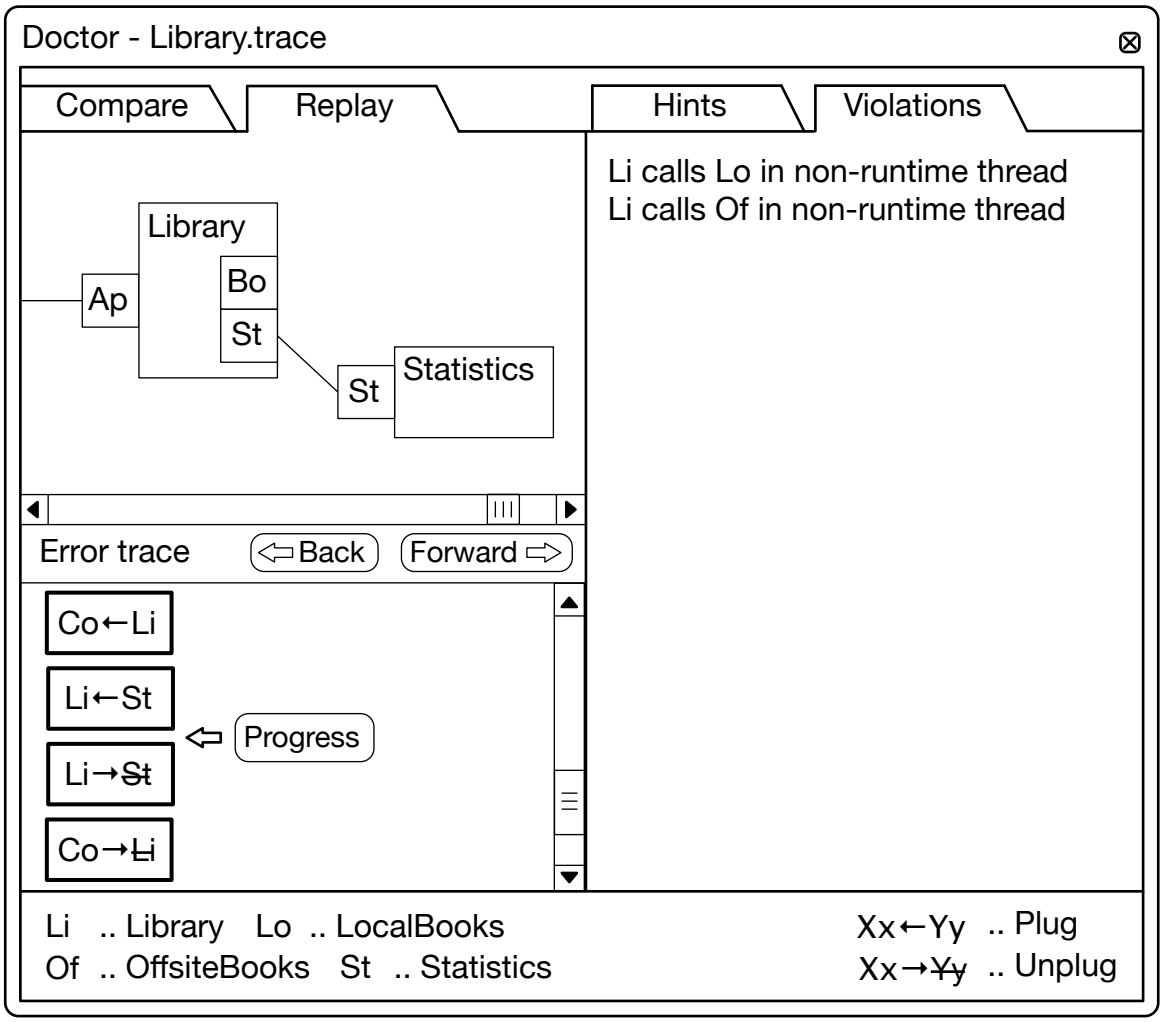


Figure 140: Composition trace replay and composition standard violations in the composition debugging tool

Chapter 7: Summary

This chapter summarizes the contributions of this thesis and discusses how they address the problem statement from Section 1.2. The thesis concludes with an outlook on future research and the current state of this work.

7.1 Contributions

Although component-based software development is used in practice for quite some time and was recently enhanced with dynamic composition, composability testing has been neglected so far. To the best of our knowledge our testing method is the first that systematically tests components for their composability, and our debugging method is the first that supports developers to locate causes for composability errors. Additionally, our classification of composition mechanisms systematically describes current component systems by their contributor provision characteristics. In detail, this thesis contributes the following:

- We propose a collection of *contributor provision characteristics* that describe how contributor components can be provided to a host component in a component system.
- We propose a collection of *composition mechanisms* that allows classifying component systems by their contributor provision characteristics.
- We propose a *composability fault classification* that describes which composability faults are typical for a given composition mechanism.
- We propose a *composability test method* that systematically tests the composability of components in dynamically composed programs.
- We propose a *composition debugging method* that analyzes composition traces of dynamically composed programs and gives hints to possible causes for composability errors.
- We provide a *composability testing tool* that automates the proposed composability test method.
- We provide a *debugging tool* that automates the proposed debugging method.

7.2 Conclusions

The problem statement in Section 1.2 stated three problems in state-of-the-art component testing methods, which make it hard to test dynamically composed programs and which are addressed by the contributions of this thesis:

- *Problem 1: Specifying test-relevant composition scenarios*
  This problem has been solved in this thesis, because the contributor provision characteristics and the composition mechanism classification from Chapter 3 allow specifying the test-relevant composition scenarios for a given composition mechanism, i.e. those scenarios that have a high probability for revealing errors. Additionally, the composability fault classification gives an overview of the possible composability faults for a given composition mechanism.
- *Problem 2: Selecting test-relevant composition scenarios*
  This problem has been solved in this thesis, because the composability test method Act generates all possible composition scenarios and selects a manageable and representative subset of them. The composability test tool Actor automates the method Act. Figure 141 shows the composability faults covered by Act.
- *Problem 3: Finding the differences between composition scenarios that show an error and those that do not*
  This problem has been solved in this thesis, because the composition debugging method Doc records, filters, splits and compares composition traces which show an error with those that do not. Additionally, it creates hints for possible error causes. The composition debugging tool Doctor automates the method Doc.

Contributor cardinality faults		Single mandatory vs. single optional	✕
		Single mandatory vs. multiple	✕
		Single optional vs. multiple	✕
Contributor availability faults	Time faults	Host instantiation time vs. later at run time	✕
		Host instantiation time vs. on notification	✕
	Order faults	Predictable order vs. unpredictable order (same contract)	✕
		Predictable order vs. unpredictable order (different contracts)	✕
		Same order on every run vs. unpredictable order (same contract)	✕
		Same order on every run vs. unpredictable order (different contracts)	✕
	Duration faults	All at once vs. continuously (same contract)	✕
		All at once vs. continuously (different contracts)	✕
Contributor identification faults			✕
Contributor instantiation faults		By host vs. by infrastructure	✕
		Globally uniform vs. host-specific	✕
Contributor registration faults		Global availability vs. host-specific availability	✕
		Global usage vs. host-specific usage	✕
Contributor sharing faults			✕
Composition standard violations		Use of an unplugged contributor	✕
		Use of not-plugged component	✕
		Contributor call in non-runtime thread	✕
		Composition state mismatch	✕

Covered

Figure 141: Composability faults covered by the composability test method Act

7.3 Future research

The contributions of this thesis allow developers to systematically test and debug dynamically composed programs. The following points could be addressed by future research in order to further simplify composability testing in practice.

1. The splitting of composition traces in the Doc method could be improved by enriching the time-based clustering with semantic information to better identify related composition operations. The comparison of composition trace parts could be improved by automatically identifying parts that are similar, e.g., parts that compose and decompose the same part of the program.

2. The Actor tool could be improved by executing the test cases in parallel on multiple computers to reduce execution time. The test cases could also be ordered in such a way that the test cases that are most likely to find errors are executed first. In that way a large number of errors could already be found early in testing, whereas executing more test cases would only increase accuracy. Finally, Actor could be integrated into a build server in order to automate the test case execution.

7.4 Current state

The composability test tool Actor and the debugging tool Doctor are publicly available together with the Plux composition infrastructure for desktop applications at *http://www.ssw.jku.at/Research/Projects/Plux/*. An ongoing dissertation project extends Plux to support distributed multi-user web applications. Other student projects port Plux, which is currently implemented in .Net, to Delphi and Java.

Appendix

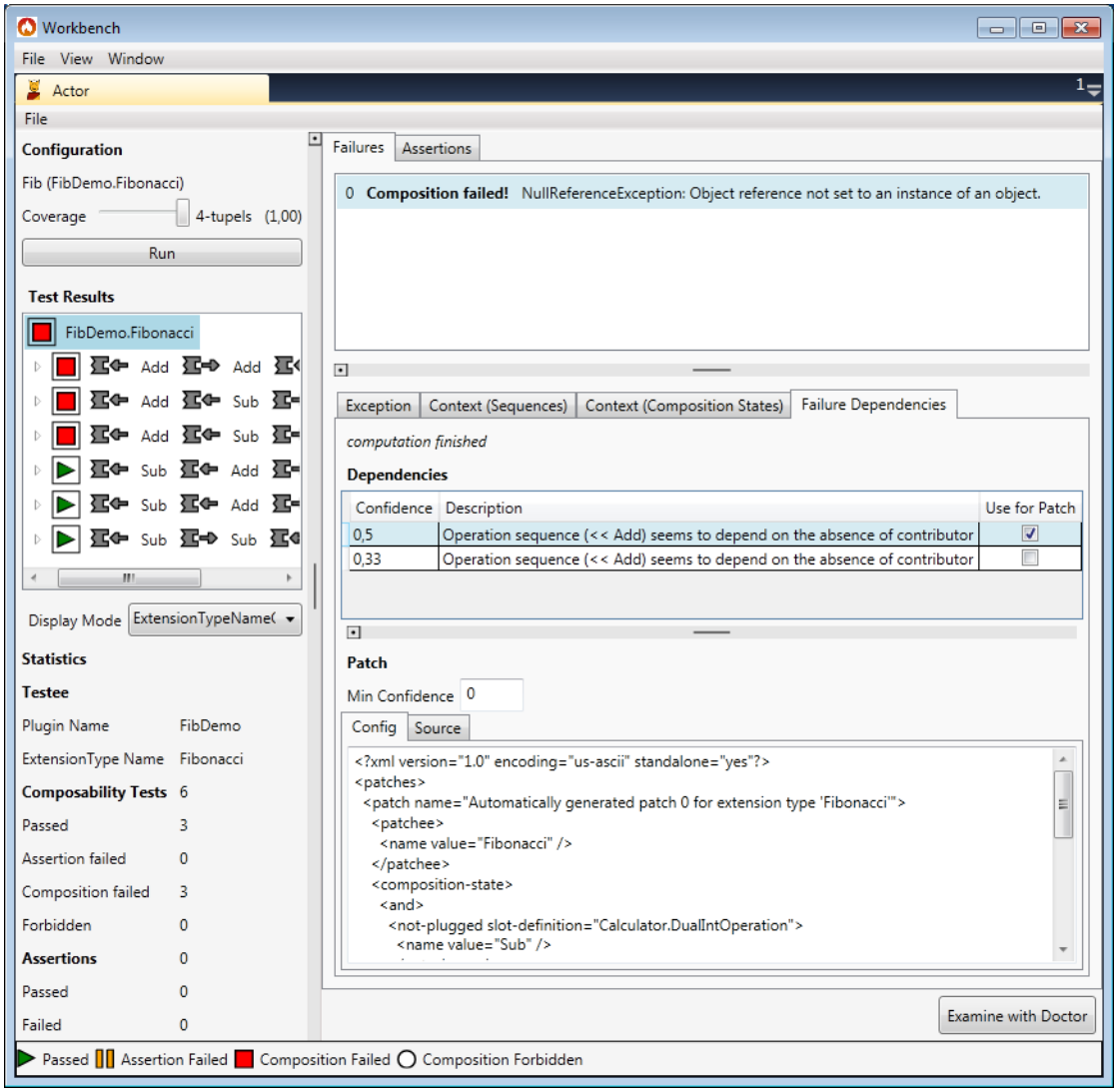


Figure 142: Screenshot of the automatic composability test tool Actor

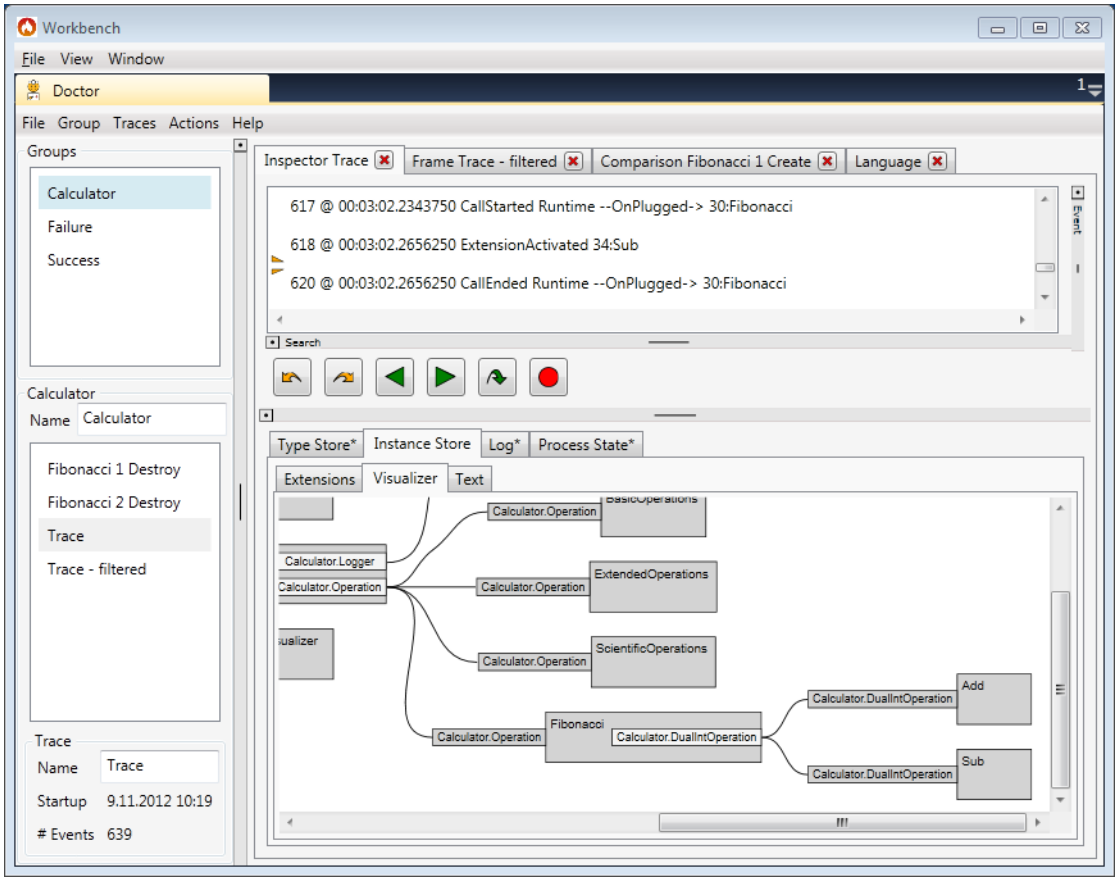


Figure 143: Screenshot of the composition debugger Doctor, replaying a composition trace

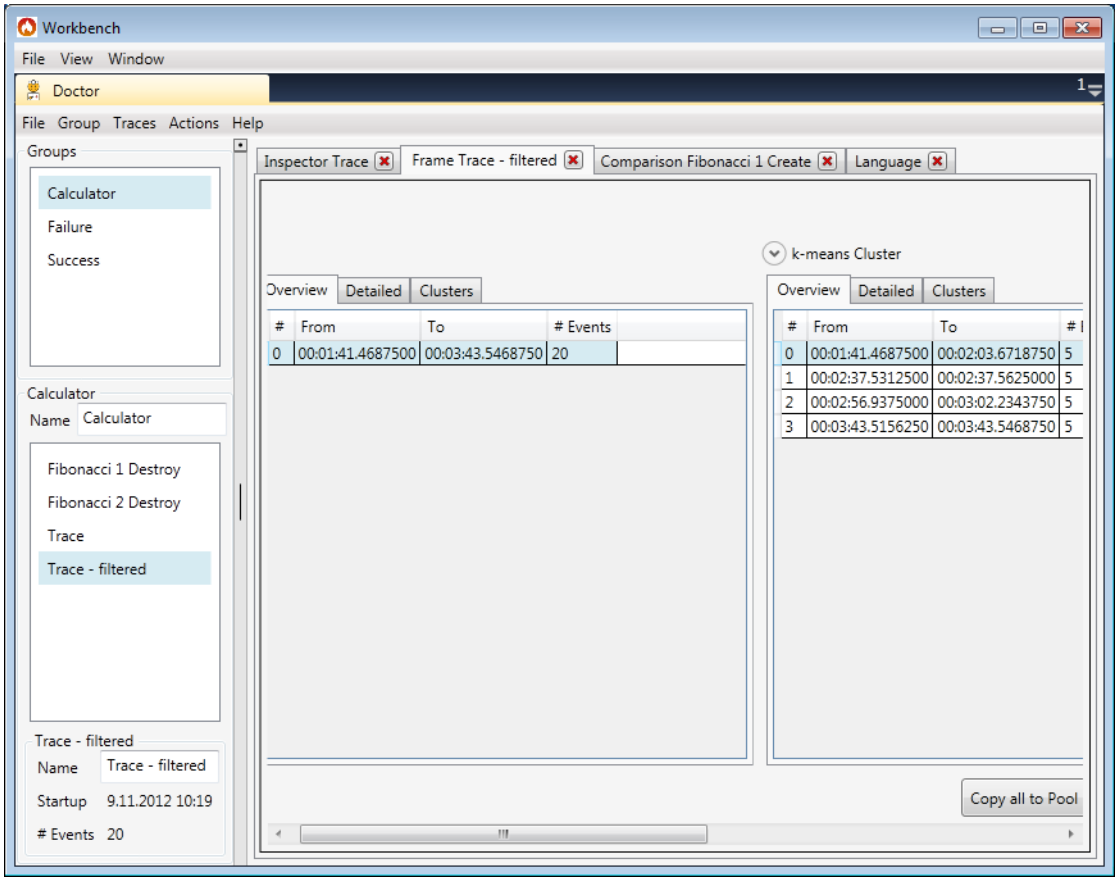


Figure 144: Screenshot of the composition debugger Doctor, splitting a composition trace

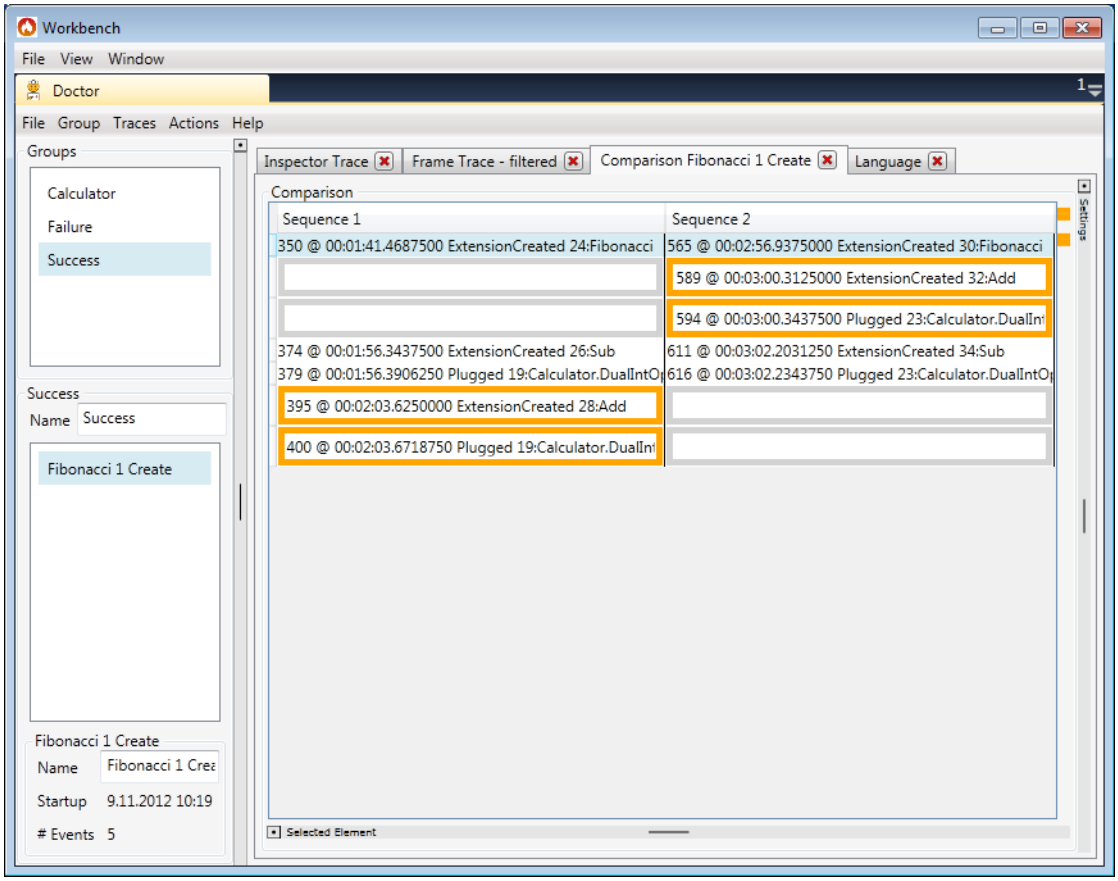


Figure 145: Screenshot of the composition debugger Doctor, comparing two composition traces

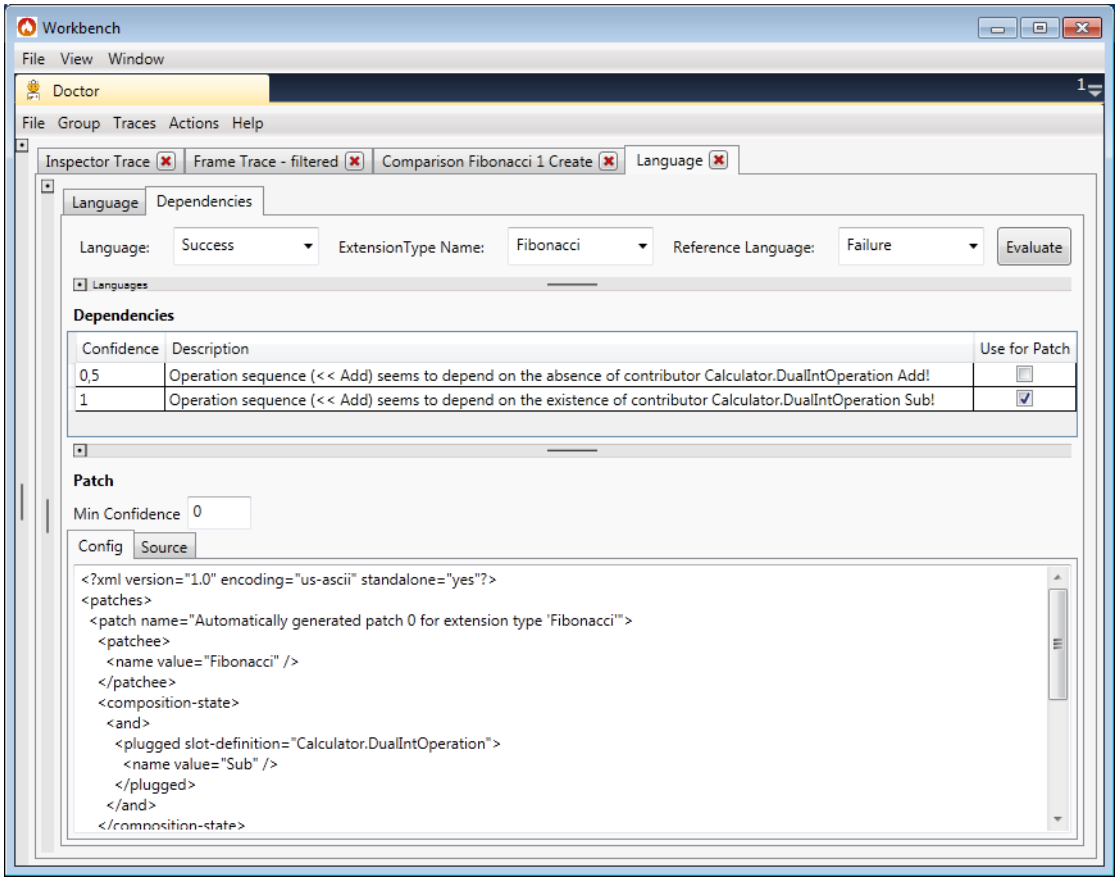


Figure 146: Screenshot of the composition debugger Doctor, reasoning error causes

List of figures

Bibliography

Abu-Eid: Testing OSGi-based Applications with DA-Testing Framework, http://www.dynamicjava.org/projects/da-testing/overview, 2009.

ANSI: *Software engineering standards,* Institute of Electrical and Electronics Engineers, 1987.

Baker,C.: Review of D.D. McCracken's "Digital Computer Programming", *Mathematical Tables and Other Aids to Computation*, 11, pp. 298-305, 1957.

Beck,K.: *Extreme Programming Explained: Embrace Change,* Addison-Wesley Professional, 1999.

Beizer,B.: *Software Testing Techniques, 2nd Edition,* Intl Thomson Computer Pr (T), 1990.

Bertolino,A., and Polini,A.: A framework for component deployment testing, *Software Engineering, 2003. Proceedings. 25th International Conference on*, IEEE, pp. 221-231, 2003.

Besson,F., Leal,P., Kon,F., Goldman,A., and Milojicic,D.: Towards automated testing of web service choreographies, *hal.archives-ouvertes.fr*, 2011.

Beust,C., and Suleiman,H.: *Next generation java™ testing: testng and advanced concepts,* Addison-Wesley Professional, 2007.

Birsan,D.: On plug-ins and extensible architectures, *Queue*, 3(2), Association for Computing Machinery, Inc, One Astor Plaza, 1515 Broadway, New York, NY, 10036-5701, USA, pp. 40-46, 2005.

Boudreau,T., Tulach,J., and Wielenga,G.: *Rich Client Programming: Plugging into the NetBeans Platform,* 1, Prentice Hall International, 2007.

da Silva,C. E., and de Lemos,R.: Dynamic plans for integration testing of self-adaptive software systems, *Proceeding of the 6th international symposium on Software engineering for adaptive and self-managing systems*, ACM, pp. 148-157, 2011.

EasyMock: EasyMock, http://easymock.org, 2012.

Eclipse: Test & Performance Tools Platform, http://www.eclipse.org/tptp, 2012.

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

ECMA: Common Language Infrastructure (CLI), 4th edn., 2006.

Eder,M.: Ein Plugin-basiertes Werkzeug zur Überwachung von Web-Inhalten, 2008.

Fröhlich,J., and Schwarzinger,M.: Improve Component-Based Programs with Connectors, *Modular Programming Languages*, Springer, pp. 306-325, 2006.

Fröhlich,J. H., and Schwarzinger,M.: Treating Interfaces as Components, Citeseer, 2005.

Gamma,E.: The extension objects pattern, *3rd Conference on Pattern Languages of Programs (PLoP'96)*, 1996.

Gao,J.: Component testability and component testing challenges, *Proceedings of International Workshop on Component-based Software Engineering (CBSE2000, held in conjunction with the 22nd International Conference on Software Engineering (ICSE2000)*, 2000.

Gelperin,D., and Hetzel,B.: The growth of software testing, *Communications of the ACM*, 31(6), ACM, pp. 687-695, 1988.

Gruber,A.: Konfigurationswerkzeug "Plugin-Explorer" für die Plugin-Plattform Plux.NET, 2010.

Hagmüller,P.: Plux for Delphy (to be published), 2012.

Hartman,A.: Software and hardware testing using combinatorial covering suites, *Graph Theory, Combinatorics and Algorithms*, Springer, pp. 237-266, 2005.

Heineman,G. T., and Councill,W. T.: *Component-based software engineering: putting the pieces together,* 17, Addison-Wesley USA, 2001.

Hewett,R., and Kijsanayothin,P.: Automated test order generation for software component integration testing, *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, IEEE Computer Society, pp. 211-220, 2009.

Hribernig,T.: Retrofitting Security in Component-based Applications (to be published), 2012.

Jahn,M.: Entwurf und Implementierung eines Cross-Compilers von Delphi nach C#, 2008.

Jahn,M., Löberbauer,M., Wolfinger,R., and Mössenböck,H.: Rule-Based Composition Behaviors in Dynamic Plug-In Systems, *Software Engineering Conference (APSEC), 2010 17th Asia Pacific*, IEEE, pp. 80-89, 2010a.

Jahn,M., Wolfinger,R., Löberbauer,M., and Mössenböck,H.: Composing user-specific web applications from distributed plug-ins, *Computer Science-Research and Development*, Springer, pp. 1-21, 2011.

Jahn,M., Wolfinger,R., and Mössenböck,H.: Extending Web Applications with Client and Server Plug-ins, *Gesellschaft für Informatik (GI)*, pp. 33, 2010b.

JMock: JMock, http://www.jmock.org, 2012.

Johnson,R., Hoeller,J., Donald,K., Sampaleanu,C., Harrop,R., Arendsen,A., Risberg,T., Davison,D., Kopylenko,D., Pollack,M., Templier,T., Vervaet,E., Tung,P., Hale,B., Colyer,A., Lewis,J., Leau,C., Fisher,M., Brannen,S., Laddad,R., Poutsma,A., Beams,C., Abedrabbo,T., Clement,A., Syer,D., Gierke,O., and Stoyanchev,R.: *Spring Reference Documentation 3.1,* 2011.

JUnit: JUnit, http://www.junit.org/, 2011.

Kernighan,B., and Ritchie,D.: *C Programming Language, The (ANSI C Version),* Prentice Hall India, 1988.

Knuth,D.: *The Art of Computer Programming, Volume 4, Generating All Tuples and Permutations, Fascicle 2,* Addison-Wesley, 2005.

Kranzlmüller,D., Löberbauer,M., Maurer,M., Schaubschläger,C., and Volkert,J.: Automatic Testing of Nondeterministic Parallel Programs, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications-Volume 2*, CSREA Press, pp. 538-544, 2002a.

Kranzlmüller,D., Maurer,M., Löberbauer,M., Schaubschläger,C., and Volkert,J.: Ant—A Testing Environment for Nondeterministic Parallel Programs, *Distributed and Parallel Systems*, Springer, pp. 125-132, 2002b.

Lengauer,P.: Trace-based Debugger for Dynamically Composed Applications, 2012.

Liu,W., and Dasiewicz,P.: Formal test requirements for component interactions, *Electrical and Computer Engineering, 1999 IEEE Canadian Conference on*, 1, IEEE, pp. 295-299 vol. 1, 1999.

Lloyd,S.: Least squares quantization in PCM, *Information Theory, IEEE Transactions on*, *28*(2), IEEE, pp. 129-137, 1982.

Löberbauer,M., Wolfinger,R., Jahn,M., and Mössenböck,H.: Testing the composability of plug-and-play components: A method for unit testing of dynamically composed applications, *Intelligent Systems and Informatics (SISY), 2010 8th International Symposium on Intelligent Systems and Informatics*, IEEE, pp. 413-418, 2010.

Löberbauer,M., Wolfinger,R., Jahn,M., and Mössenböck,H.: Composition Mechanisms Classified by their Contributor Provision Characteristics, *Intelligent Systems and Informatics (SISY), 2012 10th International Symposium on Intelligent Systems and Informatics*, IEEE, 2012.

Mariani,L., Papagiannakis,S., and Pezze,M.: Compatibility and regression testing of COTS-component-based software, *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, IEEE, pp. 85-95, 2007.

Mariani,L., and Pezze,M.: Behavior capture and test: Automated analysis of component integration, *Engineering of Complex Computer Systems, 2005. ICECCS 2005. Proceedings. 10th IEEE International Conference on*, IEEE, pp. 292-301, 2005.

McCluskey Jr.,E. J.: Minimization of Boolean functions, *Bell System Technical Journal*, 1956.

Microsoft,Equipment_Corporation,D.: *The component object model specification,* 0.9, Microsoft Corporation and Digital Equipment Corporation, 1995.

Microsoft: *Unity 2.0, General Purpose Dependency Injection Mechanism for your.Net Applications,* Microsoft Corporation, 2010a.

Microsoft: MEF Programming Guide, http://msdn.microsoft.com/en-us/library/dd460648.aspx, 2010b.

Mittermair,C.: Umstrukturierung eines monolithischen Softwaresystems in ein Plug-In-basiertes Komponentensystem, 2009.

Myers,G. J.: *The Art of Software Testing,* Wiley, 1979.

NBS: *Guideline for lifecycle validation, verification, and testing of computer software,* U.S. Dept. of Commerce, National Bureau of Standards, 1984.

Needleman,S. B., and Wunsch,C. D.: A general method applicable to the search for similarities in the amino acid sequence of two proteins, *Journal of molecular biology*, 48(3), Elsevier Science, pp. 443-453, 1970.

Oracle: ServiceLoader, http://docs.oracle.com/javase/6/docs/api/java/util/ServiceLoader.html, 2006.

OSGi: OSGi Service Platform - Release 4, http://www.osgi.org, 2011.

Petrick,S. R.: A direct determination of the irredundant forms of a Boolean function from the set of prime implicants, *Air Force Cambridge Res. Center Tech. Report*, pp. 56-110, 1956.

Petzold,C.: *Programming Windows, Fifth Edition,* Microsoft Press, 1998.

Pichler,R.: Metrix - A Measuring Tool for Run-time Figures in Plug-in based.NET Applications, 2009.

PicoContainer: Constructor Injection, http://picocontainer.org/constructor-injection.html, 2011.

PicoContainer: PicoContainer, http://www.picocontainer.org, 2012.

Plux: http://www.ssw.jku.at/Research/Projects/Plux/, 2012.

Quine,W. V.: A way to simplify truth functions, *The American Mathematical Monthly*, 62(9), JSTOR, pp. 627-631, 1955.

Reinthaler,T.: Deployment Assistant for Plux, 2012.

Reiter,S., and Wolfinger,R.: Erfahrungen bei der Portierung von Delphi Legacy Code nach.NET, *Nachwuchs-Workshop, SE*, pp. 27-30, 2007.

Saglietti,F., and Pinte,F.: Automated unit and integration testing for component-based software systems, *Proceedings of the International Workshop on Security and Dependability for Resource Constrained Embedded Systems*, ACM, pp. 5, 2010.

Schenkermayr,B.: Ein komponentenbasierter Taschenrechner auf Basis von Plux (to be published), 2012.

SWTBot: SWTBot, http://www.eclipse.org/swtbot, 2012.

Szyperski,C., Gruntz,D., and Murer,S.: *Component software: beyond object-oriented programming,* Addison-Wesley Professional, 2002.

Committee: Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification Version 1.2, *TIS Committee*, 1995.

Weinreich,R., and Sametinger,J.: Component models and component services: Concepts and principles, *Component-Based Software Engineering: Putting Pieces Together*, pp. 33-48, 2001.

Weiss,S.: Kundenbeziehungsmanagement Plux-CRM für Plux.NET, 2010.

Weyuker,E. J.: Testing component-based software: A cautionary tale, *Software, IEEE*, 15(5), IEEE, pp. 54-59, 1998.

Wolfinger,R., Dhungana,D., Prähofer,H., and Mössenböck,H.: A Component Plug-In Architecture for the. NET Platform, *Modular Programming Languages*, Springer, pp. 287-305, 2006.

Wolfinger,R., Löberbauer,M., Jahn,M., and Mössenböck,H.: Adding genericity to a plug-in framework, *ACM SIGPLAN Notices*, 46(2), ACM, pp. 93-102, 2010.

Wolfinger,R., Löberbauer,M., Jahn,M., and Mössenböck,H.: Retrofitting Security in Component-based Applications (submitted), *Computer Science-Research and Development*, Springer, 2012.

Wolfinger,R., and Prähofer,H.: Integration models in a.NET plug-in framework, *SE 2007 Conference on Software Engineering*, 2007.

Wolfinger,R.: Dynamic application composition with Plux.NET: composition model, composition infrastructure, 2010.

Wu,Y., Chen,M. H., and Offutt,J.: UML-based integration testing for component-based software, *COTS-Based Software Systems*, 2003.

Wu,Y., Pan,D., and Chen,M. H.: Techniques for testing component-based software, *Engineering of Complex Computer Systems, 2001. Proceedings. Seventh IEEE International Conference on Engineering of Complex Computer Systems*, 2001.

Zheng,J., Williams,L., and Robinson,B.: Pallino: automation to support regression test selection for COTS-based applications, *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ACM, pp. 224-233, 2007.

Curriculum Vitae

Name:	Markus Löberbauer
Date of birth:	August 22, 1977
Place of birth:	Wels, Austria
Nationality:	Austria
Contact:	Markus.Loeberbauer@jku.at | M.Loeberbauer@gmail.com

Education

2003-2012	Doctorate Degree in Technical Sciences Johannes Kepler University, Linz
1998-2003	Diploma Degree in Informatics Johannes Kepler University, Linz
1997-1998	Austrian Federal Armed Forces Hiller-Kaserne, Linz Ebelsberg
1992-1997	High school Höhere technische Bundeslehranstalt, Vöcklabruck
1988-1992	Secondary school Hauptschule Gmunden-Stadt 2, Gmunden
1984-1988	Primary school Volksschule Marienbrücke, Gmunden