UNIVERSITY OF CALIFORNIA,
IRVINE

Validation and Verification of Modular Software Applications

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Software Engineering

by

Negar Ghorbani

2022

DEDICATION

To my family and the 176 innocent passengers of the Ukraine International Airlines Flight 752, many of whom were seeking a higher education degree.

TABLE OF CONTENTS

Page

LIST OF FIGURES **vi**

LIST OF TABLES **viii**

ACKNOWLEDGMENTS **ix**

VITA **x**

ABSTRACT OF THE DISSERTATION **xii**

1 Introduction **1**
- 1.1 Java Platform Module System . . . 3
- 1.2 Android Dynamic Delivery . . . 7
- 1.3 Dissertation Structure . . . 9

2 Research Problem **11**
- 2.1 Problem Statement . . . 11
- 2.2 Research Hypothesis . . . 12
- 2.3 Thesis Statement . . . 14

3 Detection and Repair of Architectural Inconsistencies in Java **17**
- 3.1 Introduction . . . 18
- 3.2 Java Platform Module System . . . 20
  - 3.2.1 JPMS Goals and Potential Misuse . . . 20
  - 3.2.2 Understanding JPMS Modules . . . 22
- 3.3 Inconsistent Module Dependencies . . . 26
- 3.4 DARCY: Framework Overview . . . 30
  - 3.4.1 Detection . . . 31
  - 3.4.2 Repair . . . 34
- 3.5 Evaluation . . . 36
  - 3.5.1 RQ1: Pervasiveness . . . 38
  - 3.5.2 RQ2: Correctness . . . 39
  - 3.5.3 RQ3: Security . . . 40
  - 3.5.4 RQ4: Encapsulation . . . 43
  - 3.5.5 RQ5: Software Bloat . . . 46

3.5.6 RQ6: Performance 47
3.6 Threats to Validity 48
3.7 Related Work 49
3.8 Discussion 51

4 Dynamic Delivery Testing in Android 53
4.1 Introduction 54
4.2 Illustrative Example 56
4.3 Defect Model 60
4.3.1 Failed State Context Definition 62
4.3.2 Failed States 64
4.4 DELTADROID: Framework Overview 69
4.4.1 Test-Suite Analysis 69
4.4.2 Test-Suite Augmentation 72
4.5 Evaluation 76
4.5.1 Experimental Setup 77
4.5.2 RQ1: Effectiveness of Inducing Failed States 79
4.5.3 RQ2: Effectiveness of Revealing Installation Defects 82
4.5.4 RQ3: Performance 84
4.6 Threats to Validity 86
4.7 Related Work 87
4.8 Discussion 89

5 Run-time Automatic Architectural Inconsistency Resolution in Java 90
5.1 Introduction 91
5.2 Overview of Java Platform Module System 93
5.3 Defect Model: Inconsistent Module Dependencies 93
5.4 DARCY$^+$: Framework Overview 94
5.4.1 Detection 94
5.4.2 Repair 97
5.5 Evaluation 98
5.5.1 RQ1: Inconsistency resolution at Run-time 99
5.5.2 RQ2: Security 101
5.5.3 RQ3: Encapsulation 104
5.5.4 RQ4: Software Bloat 108
5.5.5 RQ5: Performance 109
5.6 Threats to Validity 110
5.7 Related Work 111
5.8 Discussion 111

6 Automated Support for Architecture-based Adaptability in Modern Java Applications 113
6.1 Introduction 114
6.2 Architecture-based Development in Java 116
6.2.1 Java Platform Module System 116

6.2.2 Illustrative Example . . . 117
6.3 ACADIA: Framework Overview . . . 119
6.3.1 ACADIA's API . . . 119
6.3.2 Static Analyzer . . . 121
6.3.3 Runtime Controller . . . 123
6.4 Application of ACADIA . . . 127
6.5 Evaluation . . . 129
6.5.1 Subject Applications . . . 130
6.5.2 RQ1: Adaptation Effectiveness . . . 132
6.5.3 RQ2: ACADIA's Performance . . . 134
6.5.4 RQ3: Overhead on Applications' Execution . . . 136
6.6 Related Work . . . 136
6.7 Discussion . . . 139

7 Conclusion **141**
7.1 Research Contributions . . . 142
7.2 Future Directions . . . 144

Bibliography **145**

LIST OF FIGURES

Page

1.1 Two example modules with their inter-dependencies 6
1.2 Installation of a new DFM on K9Mail 8
1.3 An Appium test, written in Python, for validating dynamic delivery of testing scenario of Figure 1.2 9

3.1 Three example modules with their inter-dependencies 23
3.2 A high-level overview of DARCY 31
3.3 Reflective method invocation example 32
3.4 Service loader example 34
3.5 Cumulative distribution function (CDF) of the attack surface reduction at build-time 42
3.6 Cumulative distribution function (CDF) of the Ratio of Coupling (RoC) reduction at build-time 45
3.7 Cumulative distribution function (CDF) of the Normalized Cumulative Component Dependency (NCD) reduction at build-time 45
3.8 Cumulative distribution function (CDF) of software bloat reduction at build-time . 47

4.1 Installation of a new DFM on K9Mail 57
4.2 An Appium test, written in Python, to test the installation success scenario of Figure 4.1 58
4.3 An Appium test, written in Python, to test the installation failure scenario of Figure 4.1 due to no network connection 58
4.4 An Appium test, written in Python, to test the installation failure scenario of Figure 4.1 due to insufficient storage 58
4.5 The lifecycle of a DFM installation request 60
4.6 A high-level overview of DELTADROID 70
4.7 Code snippet from K9Mail app that requests the installation of *Encryption* DFM . . 70
4.8 The results of comparing alternative approaches in revealing dynamic delivery faults over time. (a)The number of discovered faults over the elapsed time for all approaches listed in Table 4.2. (b)The Area-Under-the-Curve (AUC) of the plots in Figure 4.8a. 85

5.1 A high-level overview of DARCY$^+$ 95
5.2 Cumulative distribution function (CDF) of the attack surface reduction at build- and run-time 104

5.3 revisedCumulative distribution function (CDF) of the Ratio of Coupling (RoC) reduction at build- and run-time 107
5.4 Cumulative distribution function (CDF) of the Normalized Cumulative Component Dependency (NCD) reduction at build- and run-time 107
5.5 Cumulative distribution function (CDF) of software bloat reduction at build- and run-time 109

6.1 An example Java app implemented using JPMS 118
6.2 A high-level overview of ACADIA framework 119
6.3 An example of a reflective dependency in Java 121
6.4 An example of service dependency in Java 122
6.5 The managed subsystem after loading all modules of the example Java app (Section 6.2.2) 125
6.6 The multidimensional network representing the dynamic architecture model of the illustrative example in Section 6.2.2 after loading all modules 126
6.7 Quasar's architecture graph in module-level 130
6.8 Tascalate's architecture graph in module-level 131
6.9 Rhizomatic's architecture graph in module-level 131

LIST OF TABLES

Page

3.1 Functions describing dependencies based on module directives of JPMS 27
3.2 Subject applications 37
3.3 Identified inconsistencies and robustness results 38
3.4 Result for attack-surface reduction at build-time 41
3.5 Results for encapsulation improvement at build-time 44
3.6 Results for software-bloat reduction 46
3.7 Results for execution time 47

4.1 Subject apps 76
4.2 Testing tools' effectiveness of covering *Failed* states 80
4.3 Installation defects revealed by DELTADROID 83
4.4 Execution time of DELTADROID 85

5.1 Identified Inconsistencies and Robustness at Run-time 100
5.2 Result for Attack-Surface Reduction at Build and Run-time 103
5.3 Results for Encapsulation Improvement at Build- and Run-time 106
5.4 Results for Software-Bloat Reduction at Build- and Run-time 108
5.5 Results for Execution Time 109

6.1 Commands describing ACADIA's API 120
6.2 Results of ACADIA performing an adaptation strategy on different example scenarios of three subject Java apps 132
6.3 Results for ACADIA's execution time regarding its Static Analyzer and Runtime Controller compartments 135
6.4 Results for the overhead on the execution time of the subject apps 137

ACKNOWLEDGMENTS

First and foremost, my greatest gratitude goes to my Ph.D. advisor, Prof. Sam Malek, for his guidance, support, and inspiration. He always provided insightful feedback and ensured I had access to all the required resources for my research. I am genuinely grateful for that. I would also like to offer my sincerest gratitude to my Ph.D. co-advisor, Prof. Joshua Garcia, who mentored and supported me throughout my Ph.D. studies. I am beyond grateful for his continued support and encouragement in this journey which boosted my confidence whenever I needed it.

I would like to thank the other members of my committee, Professors Cristina Lopes and Iftekhar Ahmed, for their valuable feedback and devoting time. Throughout this journey, I gained a lot of experience in collaboration and fruitful discussions with my lab-mates, members of SEAL lab at UCI, Drs. (or soon-to-be Drs.) Reyhaneh Jabbarvand, Navid Salehnamadi, Alireza Sadeghi, Mahmoud Hammad, Forough Mehralian, and Abdulaziz Alshayban.

I owe a lot to my friends for being my family away from home throughout this journey. I am very fortunate to have all of you in my life: Alireza, Elmira, Erfan, Farima, Hojjat, Katayoun, Kimia, MaryamA, MaryamR, Mohammad, Navid, Negar, Sahand, Shaghayegh, Siavash, Sumaya, Yasaman, and many more.

Most importantly, I am forever grateful for my family. Mom, Dad, and Negin, thank you so much for your endless love and support. I know it has been a tough time to bear my absence at all moments of happiness and sorrow, and I wish I could be with you. I am who I am because of you.

Finally, I would like to express my gratitude to my best friend and husband, Arian. Thank you for being the best partner I could wish for; thank you for your help, support, and encouragement through life’s ups and downs. I am blessed beyond measure to have you in my life, and I wish I knew how to truly thank you.

VITA

Negar Ghorbani

EDUCATION

Doctor of Philosophy in Software Engineering **2022**
University of California, Irvine *Irvine, CA*

Bachelor of Science in Computer Software Engineering **2016**
Sharif University of Technology *Tehran, Iran*

RESEARCH EXPERIENCE

Graduate Research Assistant **2016–2022**
University of California, Irvine *Irvine, CA*

Applied Scientist Intern **Mar-Sep 2021**
Microsoft Data and AI *Seattle, WA*

Software Engineering Research Intern **Jun-Sep 2020**
Fujitsu Laboratories of America *Sunnyvale, CA*

TEACHING EXPERIENCE

Teaching Assistant **2016–2020**
University of California, Irvine *Irvine, CA*

PUBLICATIONS

- Negar Ghorbani, Reyhaneh Jabbarvand, Navid Salehnamadi, Joshua Garcia, and Sam Malek, ”DeltaDroid: Dynamic Delivery Testing in Android”, ACM Transactions on Software Engineering and Methodology (TOSEM) 2022.
- Alexey Svyatkovskiy, Sarah Fakhoury, Negar Ghorbani, Todd Mytkowicz, Elizabeth Dinella, Christian Bird, Jinu Jang, Neel Sundaresan, and Shuvendu K. Lahiri, ”Program Merge Conflict Resolution via Neural Transformers”, ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE) 2022
- Joshua Garcia, Ehsan Kouroshfar, Negar Ghorbani, and Sam Malek, ”Forecasting Architectural Decay from Evolutionary History”, EEE Transactions on Software Engineering (TSE) 2021

- Hamid Bagheri, Jianghao Wang, Jarod Aerts, Negar Ghorbani, and Sam Malek, ”Flair: Efficient Analysis of Android Inter-Component Vulnerabilities in Response to Incremental Changes”, Empirical Software Engineering (EMSE) 2021

- Negar Ghorbani, Joshua Garcia, and Sam Malek, ”Detection and Repair of Architectural Inconsistencies in Java”, International Conference on Software Engineering (ICSE) 2019.

- Alireza Sadeghi, Reyhaneh Jabbarvand, Negar Ghorbani, Hamid Bagheri, and Sam Malek, ”A Temporal Permission Analysis and Enforcement Framework for Android”, International Conference on Software Engineering (ICSE) 2018

- Joshua Garcia, Mahmoud Hammad, Negar Ghorbani, and Sam Malek, ”Automatic Generation of Inter-Component Communication Exploits for Android Applications”, ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE) 2017

- Reza Entezari-Maleki, Seyed Ehsan Etesami, Negar Ghorbani, Arian Akhavan Niaki, Leonel Sousa, and Ali Movaghar, ”Modeling and Evaluation of Service Composition in Commercial Multi-Clouds using Timed Colored Petri Nets”, IEEE Transactions on Systems, Man, and Cybernetics 2017

ABSTRACT OF THE DISSERTATION

Validation and Verification of Modular Software Applications

By

Negar Ghorbani

Doctor of Philosophy in Software Engineering

University of California, Irvine, 2022

Professor Sam Malek, Co-Chair
Assistant Professor Joshua Garcia, Co-Chair

Modular software applications are developed based on a software design technique that emphasizes separating the program's functionality into independent, interchangeable components. Each component contains everything necessary to provide certain functionality. Building software systems using modules introduces many benefits, including improving software quality, easier maintainability, and better scalability, especially for large, complex, and long-running software systems. Although a software system's architecture is often conceptualized in terms of high-level constructs, e.g., software components, connectors, and interfaces, programming languages usually provide low-level constructs, e.g., classes, methods, and variables. To bridge this gap, many widely used programming languages, such as Java and C++, have recently incorporated the notion of module or component. The increasing support of modularization in the modern programming languages, although offering many benefits, introduces plenty of new challenges in all phases of software development and execution, including at build-time, module installation-time, and run-time. Hence, the need for effective validation and verification techniques has increased more than ever.

This dissertation proposes to advance validation and verification of modular applications through investigating the static and dynamic behavior of modular applications and introducing automatic validation, and verification techniques to (1) detect and repair architectural inconsistencies at both build- and run-time, and (2) test dynamic delivery failures at module installation-time.

All conducted experiments on real-world subject apps corroborate the effectiveness and efficiency of the proposed approaches and their ability to validate and verify the behavior of modular software applications at all different phases of their development and execution.

Chapter 1

Introduction

A software system's architecture comprises the principal design decisions employed in the system's construction [194]. The architecture of a system is often conceptualized in terms of high-level constructs, such as software components, connectors, and their interfaces. However, programming languages usually provide low-level constructs, such as classes, methods, and variables. To that end, the software engineering research community has previously advocated for *architecture-based development*, whereby a programming language (e.g., ArchJava [57]) or a framework (e.g., C2 [195]) provides the implementation constructs for realizing the architectural abstractions.

Research over the past decade has revealed that modeling software architecture at the level of components and connectors is useful in a growing variety of contexts. This has led to the development of a plethora of notations for representing software architectures, each focusing on different aspects of the systems being modeled, i.e., Architecture Description Languages (ADLs)[93].

In spite of this prior work in the academic community, until recently, most popular programming languages lacked extensive support for architecture-based development. However, this has changed with many widely used programming languages, e.g., Java, and C++, incorporating the notion of modules or components representing an architectural construct.

One example of newly introduced modularization construct is Java Platform Module System (JPMS) in Java. Java, arguably the most popular programming language over the past two decades, has introduced JPMS in its 9th version. JPMS makes it easier for developers to construct large applications and improve the encapsulation, security, and maintainability of Java applications in general as well as the JDK itself [7]. Using Java's module system, the developer explicitly specifies the system's components (i.e., modules in Java) as well as the specific nature of their dependencies. Developers can also determine the level of access granted to each module and determine which modules should be packaged together for deployment.

As another example of modern modularization constructs, consider the notion of App Bundle in Android. Android, as the leading mobile operating system worldwide [32], has recently introduced a new publishing format, called *Android App Bundle* [29]. One of the main goals of app bundles is to enable *Dynamic Delivery* [46], i.e., enhance modularization so that developers can customize their apps based on user requirements and deliver optional features, as separate modules, on a user's demand. To that end, each app bundle consists of several cohesive modules that together implement an app's full functionality. Specifically, the *base module* of the app bundle implements the core functionality of the app, *configuration modules* include device-specific resources, and *Dynamic Feature Modules* (DFMs) implement optional features. When users install an app for the first time, they, in fact, install only the base module and a configuration module specific to their devices. DFMs can be downloaded later based on the user's demand.[1]

The support of modularization in programming languages help developers with architecture-based development, hence, improved software quality, easier maintainability, and better scalability. Despite all the benefits, it introduces new challenges in different phases of development and execution of the modular software application, including build-time, module installation-time, and run-time.

Regarding the build-time, for instance, developers may introduce architectural inconsistencies while

[1]Developers define how and when users can download different Dynamic Features on devices running Android 5.0 (API level 21) or higher.

defining the modules and their dependencies. These inconsistencies can have severe security and performance consequences. On the other hand, software applications can face different challenges during the installation of modules. A module installation request can fail due to various reasons, leading to unexpected behavior or run-time exceptions in software applications. Furthermore, new modules could be introduced, loaded, and unloaded during the software applications' run-time. Loading and unloading modules define new dependencies or modify the existing dependencies. Hence, the dynamic architecture of software applications will change, and it may lead to dynamic architectural inconsistencies.

To address these challenges, first, I have formally defined defect models, describing the undesired implementations and behaviors of modular software applications in different phases of their development and execution. Then, using the defect models, I have developed validation and verification approaches that detect the undesired behaviors in software applications, properly test them under various relevant conditions, or repair them. In the remainder of this section, I will explain the related background and challenges of modular software applications in more detail.

1.1 Java Platform Module System

JPMS enables the specification of a system's architecture in terms of key architectural elements—specifically components in the form of Java modules, architectural interfaces, and resulting dependencies among components. JPMS aims to enable reliable configuration, stronger encapsulation, and modularity of the Java Development Kit (JDK) and Java Runtime Environment (JRE) to solve the problems faced by engineers when developing and deploying Java applications [185].

JPMS provides software designers and developers with strong encapsulation in their systems by modularizing and allowing explicit specification of interfaces and dependencies. Software architects or developers can specify which of a module's public types are accessible or inaccessible to other modules [177]. The more refined accessibility control in JPMS reduces the points at which a Java application may be susceptible to security attacks leading to more elegant and logical architecture designs[96].

The Java platform used to be a monolith consisting of a massive number of packages, making it challenging to develop, maintain, and evolve. With the introduction of JPMS, the Java platform is now modularized into 95 modules. As a result, JPMS allows software developers could easily choose a subset of the JDK as a platform for their applications. This significantly reduces the potential of software bloat and points of attack for malicious agents.

Using JPMS, Java developers can create lightweight custom JREs consisting of only modules they need for their application or the devices they target. As a result, the Java platform can more easily scale down to small devices, which is important for microservices or IoT devices [97].

Although JPMS allows for the specification of systems' architectures, the implementation of a Java application may be inconsistent with the specified architecture. Such inconsistencies may arise at both build- and run-time. Build-time architectural inconsistencies occur due to architects' or developers' misunderstanding of a software system's architecture (e.g., an architect mistakenly specifies a more accessible interface than he intended), or simply due to mistaken implementations (e.g., a developer neglects to use a module's interface, even though the architect intended such a use). JPMS also allows developers to modify Java applications at run-time. More specifically, developers can dynamically load or unload modules into a running Java application [153]. Loading and unloading modules at run-time change the system's architecture and may introduce new architectural inconsistencies at run-time.

Architectural inconsistencies can result in (1) a poorly encapsulated architecture, making an application harder to understand and maintain; (2) bloated software; or (3) insecure software. In terms of security, for instance, one of the potential problems is granting unnecessary access to internal classes and packages, potentially resulting in security vulnerabilities. In terms of software bloat, inconsistent dependencies can compromise the scalability and performance of Java software (e.g., requiring many unnecessary modules from the JDK).

In JPMS, a *module* is a uniquely named, reusable group of related packages, as well as resources (such as images and XML files)[7]. Each module has a descriptor file, `module-info.java`, which contains meta-data, including the declaration of a named module. A named module should specify (1) its dependencies on other modules, i.e.a, the classes and interfaces that the module needs or expects, and should specify (2) which of its own packages, classes, and interfaces are exposed to other modules.

The module declaration file consists of a unique module name and a module body. Any module body can be empty or contain one or more module directives, which specify a module's exposure to other modules or the modules it needs access to.

Figure 1.1 shows an example of a project with two modules: `foo` and `bar`. Figure 1.1a and 1.1b describe the declaration of modules `foo` and `bar`, respectively, provided in their `module-info.java` file. A module declaration can utilize combinations of five module directives [119], which specify module interfaces and their usage: the *requires* directive specifies the packages that a module needs access to, the *exports* and *opens* directives make packages of a module available to other modules, the *provides* directive specifies the services a module provides, and the *uses* directive specifies the services a package consumes.

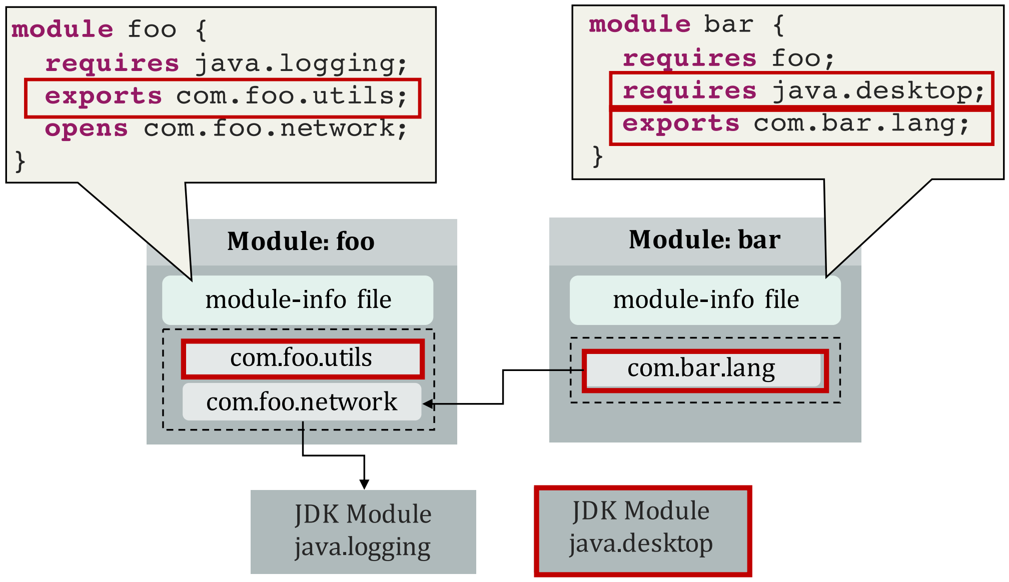
Figure 1.1c is a diagram that depicts both the specified and actual dependencies of modules `foo` and `bar` within the project based on their declarations and implementations. The arrows between the modules indicate the actual dependencies extracted from the implementations of packages and classes of the modules. The specified dependencies marked by red rectangles are excess dependencies where the module either exposes more of its internals than are used (with *exports*) or requires internals of other modules that it never uses (with *requires*). These excess dependencies are examples of architectural inconsistencies which can result in security threats and reduce the

```
module foo {
    requires java.logging;

    exports com.example.foo.utils;

    opens com.example.foo.network;
}
```

(a) The declaration of module `foo`.

```
module bar {
    requires foo;

    requires java.desktop;

    exports com.bar.lang;
}
```

(b) The declaration of module `bar`.



```
module foo {
  requires java.logging;
  exports com.foo.utils;
  opens com.foo.network;
}
```

```
module bar {
  requires foo;
  requires java.desktop;
  exports com.bar.lang;
}
```

(c) The architectural inconsistencies within the `foo` and `bar` modules.

Figure 1.1: Two example modules with their inter-dependencies

performance and maintainability of the software application.

First, to resolve architectural inconsistencies, all the inconsistencies between the specified and implemented dependencies must be detected at both build- and run-time. Then, to repair the inconsistencies, each module's specification should be modified in a way that makes it consistent with the module's implemented dependencies. Manually detecting and repairing architectural inconsistencies requires detailed knowledge of the system's architecture, the system's specified and implemented dependencies, and their differences. Therefore, it is a highly time-consuming task and may be inaccurate. Chapters 3 and 5 describe the details of our approach for automatic detection and repair of architectural inconsistencies at build- and run-time, respectively.

c

1.2 Android Dynamic Delivery

Android app bundle, as a new publishing format, consists of several cohesive modules, including the *base module* implementing the core functionalities of the app, several *configuration modules* containing device-specific resources, and several *Dynamic Feature Modules* (DFMs) implementing optional features of the app. The main goal of Android app bundle is to enable *Dynamic Delivery* [46], where DFMs can be downloaded and installed on the fly, upon users' demands. When users install an app for the first time, they, in fact, install only the base module and a configuration module specific to their devices, which significantly reduces the initial size of the app.

To install a DFM, i.e., an optional feature, the app's base module should send an *installation request* to Google Play through Play Core API (the app's runtime interface with Google Play). Once Google Play accepts the request, the app downloads the requested DFM and installs it on the user's device. The lifecycle of an installation request is not always straightforward, and, in fact, such requests may *fail* due to several reasons. As a result, developers should monitor the state of an installation request from start to finish and adjust the behavior of an app if a failure happens.

As an illustrative example, I use an Android app called K9Mail [31]. In this subsection, I briefly describe the functionality of K9Mail, a defect in the app associated with dynamic delivery, and how a test can reveal the defect.

App: K9Mail is an email client app whose main functionalities are viewing and sending emails as well as managing email accounts. In addition to these core functionalities, K9Mail offers an optional feature for encrypting emails to provide an additional level of security. Since most users do not use this feature, it is encapsulated into a DFM that can be installed on-demand by users.

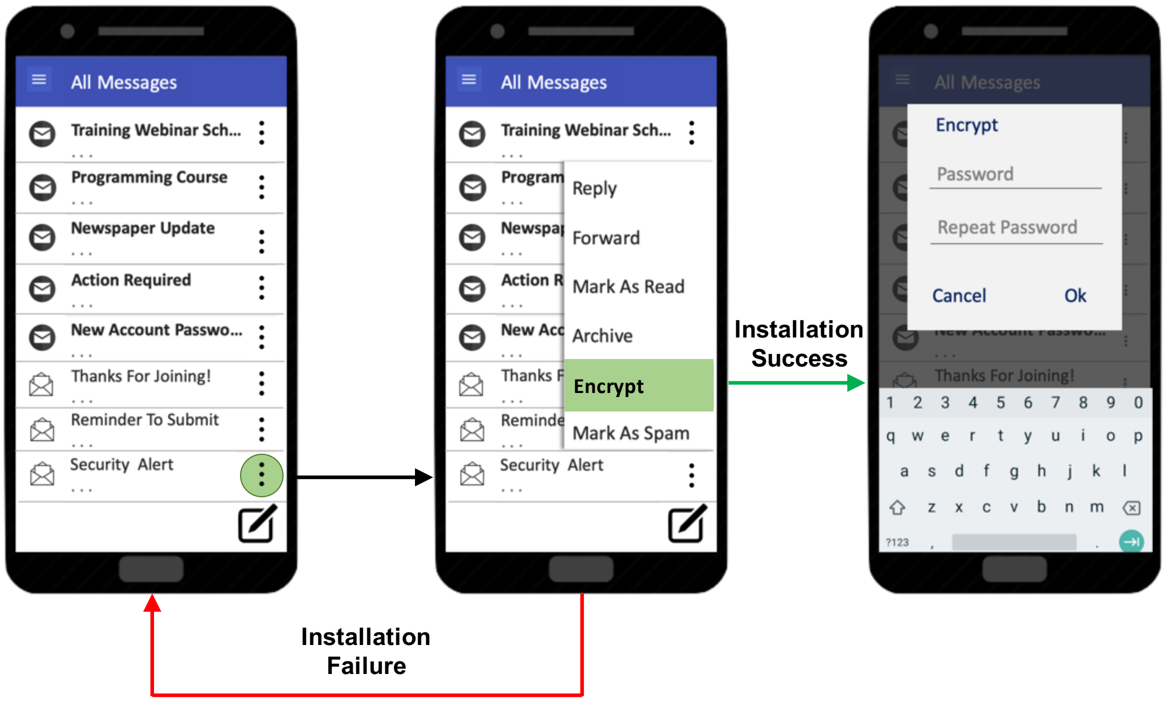


Figure 1.2: Installation of a new DFM on K9Mail

Defect Scenario: Figure 1.2 shows three steps that lead to installation of the *Encryption* DFM. First, the user clicks on an email's menu—denoted by three dots—to access more options. Clicking on the "Encrypt" option on the menu initiates the installation of the *Encryption* DFM. If the module installation is successful, the user will see a dialog to enter the password required for the encryption key and confirms the encryption. Otherwise, if the installation request fails for any reason (see Section 4.3), e.g., there is no network connection available to send the request, K9Mail goes to the initial screen without notifying the users about the failure or instructing them to take proper actions—potentially giving the user the false impression that the email is encrypted. A developer may not detect this defect without a proper test to reveal the failure scenario.

Test: Figure 1.3 shows an Appium [40] system test that validates the behavior of K9Mail during installation of the *Encryption* module when there is no network signal. Any network error [25] that happens while DFM installation is in process, i.e., an app is downloading or installing a DFM, will cause the installation to fail (see Section 4.3). As a result, the test disconnects the network

```python
#click on the "Security Alert" email's menu icon
driver.find_element_by_xpath(path_to_icon).click()
#click on the "Encrypt" option
driver.find_element_by_xpath(path_to_encrypt).click()
#enter encryption password in the password field
driver.find_element_by_id(password_field_ID).
     send_keys(password)
#repeat password
driver.find_element_by_id(password_confirm_field_ID).send_keys(password)
#click on "OK" button
driver.find_element_by_id(OK_button_ID).click()
```

Figure 1.3: An Appium test, written in Python, for validating dynamic delivery of testing scenario of Figure 1.2

connection before clicking on the “Encrypt” option, making the installation request fail.

There are two main challenges in designing such tests: (1) Installation request failures depend on the contextual settings in which a test is executed and cannot be achieved simply through GUI actions. (2) The position of the injected events in the test event sequence is critical. If the test disables the network connection at a wrong location in the test events, the app's behavior might change and undermine the test as a whole. Chapter 4 describes the above-mentioned example and our automatic approach for dynamic delivery testing of Android in more detail.

1.3 Dissertation Structure

The rest of this dissertation is organized as follows. Chapter 2 presents the research problem, the thesis statement, and three research hypotheses. Chapter 3 explains the architectural inconsistencies in Java applications in-depth and presents DARCY, an automatic technique for detection and repair of architectural inconsistencies in Java at build-time. Chapter 4 describes dynamic delivery failures in Android applications and explains the details of DELTADROID, an automatic technique for testing dynamic delivery in Android applications. Chapter 5 addresses the run-time architectural inconsistencies and introduces DARCY$^+$ that detects and repairs architectural inconsistencies not only statically at build-time but also dynamically at run-time, which may arise due to dynamic changes occurring to software systems at run-time. Chapter 6 explores the adaptation capabilities of modularized Java applications, inspired by their run-time behavior, and presents ACADIA, a framework that allows any Java 9+ software system to be adapted architecturally at run-time, followed by a conclusion in Chapter 7.

The following is the list of my research projects and publications presented in this dissertation:

- Negar Ghorbani, Joshua Garcia, and Sam Malek, ”Detection and Repair of Architectural Inconsistencies in Java,” 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE) 2019.
- Negar Ghorbani, Reyhaneh Jabbarvand, Navid Salehnamadi, Joshua Garcia, and Sam Malek, ”DELTADROID: Dynamic Delivery Testing in Android,” ACM Transactions on Software Engineering and Methodology (TOSEM), 2022.
- Negar Ghorbani, Tarandeep Singh, Joshua Garcia, and Sam Malek, ”Darcy: Automatic Architectural Inconsistency Resolution in Java,” manuscript ready for submission.
- Negar Ghorbani, Joshua Garcia, and Sam Malek, ”Automated Support for Architecture-based Adaptability in Modern Java Applications,” manuscript ready for submission.

In addition, the following publications are not included in the dissertation but are related:

- Joshua Garcia, Ehsan Kouroshfar, Negar Ghorbani, and Sam Malek, ”Forecasting Architectural Decay from Evolutionary History,” IEEE Transactions on Software Engineering (TSE) 2021.
- Hamid Bagheri, Jianghao Wang, Jarod Aerts, Negar Ghorbani, and Sam Malek, ”Flair: Efficient Analysis of Android Inter-Component Vulnerabilities in Response to Incremental Changes,” Empirical Software Engineering (EMSE) 2021.

Chapter 2

Research Problem

2.1 Problem Statement

The increasing support of modularization in modern programming languages, although offering many benefits, introduces plenty of new challenges in all phases of software development. Software developers need to make sure that they are defining and using the modules correctly without creating additional errors or vulnerabilities. They should also have appropriate tests to assess the modules' behavior throughout the software's execution. Hence, the need for effective validation and verification techniques has increased more than ever. The challenges caused by the lack of validation and verification techniques to effectively and efficiently analyze and test modular software applications can be summarized as follow:

"The rising support of modular software constructs by modern programming languages has motivated the need to further investigate the static and dynamic behavior of software applications under the impact of multiple modules being installed or uninstalled. More specifically, there is a demand for practical techniques that ensure the architectural consistency between the designed application and the implemented one at both build- and run-time, which further protects and optimizes the

application in various architectural metrics. Additionally, effective testing and analysis techniques are required to assess the behavior of modular applications under different conditions, both at the module installation- and run-time. However, unfortunately, there is a lack of appropriate frameworks and techniques among state-of-the-art and state-of-the-practice that can effectively ensure the quality of the modular software application leveraging both static and dynamic program analysis techniques."

2.2 Research Hypothesis

This research investigates the following hypotheses:

Before developing modular software applications, developers need to identify the system's components (i.e., modules) and the nature of their dependencies. This will form the system's *prescriptive architecture*. Programming languages supporting modular constructs allow developers to define the system's prescriptive architecture. For instance, using Java's module system, developers explicitly specify Java modules and all their dependencies in a file called `module-info.java`.

The prescriptive architecture should match the architecture reflected in the system's implementation, known as the *descriptive architecture*. Inconsistencies between prescriptive and descriptive architectures are of utmost concern in any software project since architecture is the primary determinant of a software system's key properties. However, programming languages do not necessarily provide mechanisms to ensure the specified prescriptive architecture is, in fact, consistent with the descriptive architecture of the implemented software. For instance, in the case of modular Java applications, Java does not validate whether the declared dependencies in the `module-info` file are accurately reflecting the implemented dependencies among the system's components.

Build tools use the module declarations to determine the level of access granted to each module and which modules should be packaged together for deployment. As a result, inconsistencies between prescriptive and descriptive architecture in modular applications have severe security and performance consequences. These inconsistencies also affect the engineers' ability to use the prescriptive architecture to understand the system's properties or to make maintenance decisions.

Hypothesis 1: *An automated and efficient static analysis framework can be devised that accurately detects and robustly repairs the architectural inconsistencies within a modular software application.*

One of the many benefits of modular programming is that modules can be installed or uninstalled throughout the execution of the software application. This decision can be based on user demands, dynamic storage consumption, or optimization measures. The lifecycle of a module installation process is not always straightforward, and, in fact, module installations may fail due to many reasons. As a result, developers should monitor the state of the software applications during the module installation process from start to finish and adjust the application's behavior if a failure happens.

To ensure the proper behavior of software applications during the installation of a module, developers should have tests that initiate all module installations in the application and verify the app's behavior under any condition that can induce a module installation failure. The challenge here lies in the fact that such failures happen under peculiar conditions, e.g., environmental and contextual situations. Since these conditions can be out of the scope of the applications and cannot be effectively produced through GUI actions, most existing testing techniques that target GUI testing are unsuitable for such a purpose.

Hypothesis 2: *A testing technique that considers the module installation lifecycle and their fault models can be developed to effectively test the behavior of software applications during module installations.*

The capability of modules being loaded or unloaded dynamically at run-time impacts the software systems' architecture. The dynamic nature of a modular application might introduce potential architectural inconsistencies that can impair the systems' security, maintainability, and memory consumption. To address this challenge, developers should constantly monitor the state of software applications during their run-time to analyze the systems' dynamic architecture and ensure the conformance between the systems' intended and implemented architecture.

Hypothesis 3: *An automated and efficient framework can be devised that continuously monitor the dynamic architecture of modular software applications at run-time, accurately detect any emerging architectural inconsistencies, and robustly repair them in real-time with low overhead.*

2.3 Thesis Statement

The problem statement described in Section 2.1 and the key insights described in Section 2.2 motivate the need for novel techniques and tools that (1) accurately detect and robustly repair the architectural module inconsistencies at both build- and run-time, and (2) automatically test module installation failures with high coverage fault detection.

To that end, regarding architectural inconsistencies, I defined a novel defect model consisting of 8 types of architectural inconsistencies in Java 9+ applications. Moreover, I propose DARCY and DARCY^{+}, automated frameworks that leverage the defect model and custom static analyses to automatically detect and repair the identified inconsistencies within Java applications both statically at build-time and dynamically at run-time. More specifically, I developed a custom static analysis that identifies different types of package dependencies, including the reflective dependencies. The static analysis approach uses a backwards analysis that follows the use-def chain of the corresponding invoked methods, e.g., reflective method of java.lang.reflect.Method.

Furthermore, regarding testing module installation failures, I propose DELTADROID, an approach for dynamic delivery testing of Android. DELTADROID is a test-suite augmentation approach that leverages static and dynamic analysis techniques to (1) identify tests that reach Dynamic Feature Modules (DFMs) installation requests and (2) modify the identified tests with a combination of GUI and system actions to generate new test cases that induce different conditions, which may cause a DFM installation request to fail.

Overall, I can state my thesis statement as follows.

This dissertation improves the state-of-the-art in verification and validation of modular software systems by presenting (1) novel defect models for Android and Java 9+ applications that represent the complications caused by modularization in these platforms, (2) the first automated approach that detects and repairs the modular inconsistencies with 100% accuracy in the case of Java 9+ applications at both build- and run-time, and (3) the first automated approach that tests module installation failures in the case of Android applications with the highest fault coverage known so far.

A variety of techniques have previously focused on bridging the gap between the prescriptive and descriptive architecture of software systems with different strategies, including focusing only on the descriptive architecture [105, 140, 150, 133, 117, 114, 165, 183, 90, 175, 187, 160, 105, 140, 71], obtaining the descriptive and prescriptive architecture and checking its conformance with the prescriptive architecture [168, 161, 201, 169, 197, 55, 219, 180, 100, 182, 142, 142, 86, 141, 124, 77], providing architectural constructs in code [57, 79, 121, 154, 59, 176, 186, 198], and detecting and fixing different types of dependency conflicts [206, 208, 145, 207, 205, 204, 148]. Despite other state-of-the-art techniques, DARCY and DARCY$^+$ are the first approaches that support architectural-implementation conformance, at both build- and run-time, and include repairing non-conforming architectures rather than just determining inconsistencies. In addition, DARCY and DARCY$^+$ are the only approaches for architecture-implementation mapping that focuses on software bloat and attack-surface reduction. I manually checked the correctness of DARCY and DARCY$^+$'s detection of architectural inconsistencies, and all the discovered inconsistencies were correct, i.e., 100% accurate. I also manually verified the robustness of DARCY and DARCY$^+$'s repairs, i.e., repairing without introducing any fault or unexpected behavior, as well as ensuring the same test passing rate before and after the repairs.

Regarding testing module installation failures, I evaluated DELTADROID's fault coverage by comparing it against a baseline static analysis approach as well as random and model-based state-of-the-art and state-of-the-practice testing techniques [39, 123, 50]. DELTADROID resulted in the highest coverage in inducing installation failures and revealed the most number of installation failures among all other approaches.

Chapter 3

Detection and Repair of Architectural Inconsistencies in Java

Java is one of the most widely used programming languages. However, the absence of explicit support for architectural constructs, such as software components, in the programming language itself has prevented software developers from achieving the many benefits that come with architecture-based development. To address this issue, Java 9 has introduced the Java Platform Module System (JPMS), resulting in the first instance of encapsulation of modules with rich software architectural interfaces added to a mainstream programming language. The primary goal of JPMS is to construct and maintain large applications efficiently—as well as improve the encapsulation, security, and maintainability of Java applications in general and the JDK itself. A challenge, however, is that module declarations do not necessarily reflect the actual usage of modules in an application, allowing developers to mistakenly specify inconsistent dependencies among the modules. In this chapter, I formally define 8 inconsistent modular dependencies that may arise in Java-9 applications. I also present DARCY, an approach that leverages these definitions and static program analyses to automatically (1) detect the specified inconsistent dependencies within Java applications and (2) repair those identified inconsistencies. The results of our experiments, conducted over 38 open-source Java-9 applications, indicate that architectural inconsistencies are widespread and demonstrate the benefits of DARCY in automated detection and repair of these inconsistencies.

3.1 Introduction

A software system's architecture comprises the principal design decisions employed in the system's construction [194]. Although every system has an architecture, the architecture of many systems is not explicitly documented, for instance in the form of UML models. Ensuring that the architecture as documented or intended, known as the *prescriptive* architecture, matches the architecture reflected in the system's implementation, known as the *descriptive* architecture, remains a major challenge [194]. The architecture of a system is often conceptualized in terms of high-level constructs, such as software components, connectors, and their interfaces, while programming languages provide low-level constructs, such as classes, methods, and variables, making it a non-trivial task to map one to the other.

Inconsistencies between prescriptive and descriptive architectures are of utmost concern in any software project since architecture is the primary determinant of a software system's key properties. One promising approach for abating the occurrence of architectural inconsistencies is to make it easier to bridge the gap between architectural abstractions and their implementation counterparts. To that end, the software engineering research community has previously advocated for *architecture-based development*, whereby a programming language (e.g., ArchJava [57]) or a framework (e.g., C2 [195]) provides the implementation constructs for realizing the architectural abstractions.

In spite of this prior work in the academic community, until recently, Java—arguably one of the most popular programming languages over the past two decades—lacked extensive support for architecture-based development. This all changed with the introduction of Java Platform

Module Systems (JPMS) in Java 9 and subsequent versions[1]. Modules are intended to make it easier for developers to construct large applications and improve the encapsulation, security, and maintainability of Java applications in general as well as the JDK itself [7].

Using Java's module system, the developer explicitly specifies the system's components (i.e., modules in Java) as well as the specific nature of their dependencies in a file called `module-info`. However, Java does not provide any mechanism to ensure the prescriptive architecture specified in the `module-info` file is, in fact, consistent with the descriptive architecture of the implemented software, i.e., whether the declared dependencies in the `module-info` file are accurately reflecting the implemented dependencies among the system's components. Inconsistencies between the prescriptive and descriptive architectures in Java 9+ matter. The Java platform uses the `module-info` file to determine the level of access granted to each module and which modules should be packaged together for deployment. As a result, inconsistencies between prescriptive and descriptive architecture in Java have severe security and performance consequences. These inconsistencies also affect the engineers' ability to use the prescriptive architecture to understand the system's properties or to make maintenance decisions.

In this chapter, we formally define 8 types of modular inconsistencies that may occur in Java 9+ applications. we present DARCY, an approach that leverages these definitions and static analyses to automatically (1) detect the specified inconsistencies within Java 9+ applications at their build-time and (2) repair them. DARCY is also publicly available [52].

The results of our experiments, conducted over 52 open-source Java 9+ applications, indicate that architectural inconsistencies are widespread and demonstrate the benefits of DARCY in automated detection and repair of these inconsistencies. DARCY found 567 instances of inconsistencies at build-time among 38 Java applications in our dataset. By automatically fixing these inconsistencies, DARCY was able to measurably improve various attributes of the subject applications' architectures by reducing the attack surface of applications by 56.37%, improving their encapsulation by 24.23%,

[1]We may refer to Java 9 and subsequent versions as Java 9+ in this paper.

and producing deployable applications that consume 16.45% less memory.

The remainder of this chapter is organized as follows. Section 3.2 introduces the module system of Java and its design goals. Section 3.3 presents a defect model consisting of the architectural inconsistencies in the context of Java 9+. Section 3.4 provides details of our approach and its implementation. Section 3.5 presents the experimental evaluation of the research. Section 3.6 includes the threats to validity of our approach. The chapter concludes with an outline of related research and future work.

3.2 Java Platform Module System

To aid the reader with understanding architectural specification in Java, we introduce the new module system, called Java Platform Module System (JPMS). We overview JPMS's goals and the architectural risks that arise from its misuse. We then discuss the details of modules in Java—including module declarations and module directives.

3.2.1 JPMS Goals and Potential Misuse

JPMS enables specification of a prescriptive architecture in terms of key architectural elements—specifically components in the form of Java modules, architectural interfaces, and resulting dependencies among components. JPMS aims to enable reliable configuration, stronger encapsulation, modularity of the Java Development Kit (JDK) and Java Runtime Environment (JRE) to solve the problems faced by engineers when developing and deploying Java applications [185].

Software designers and developers can achieve strong encapsulation in their Java-9 systems by modularizing them and allowing explicit specification of interfaces and dependencies. Encapsulation in JPMS is achieved by allowing architects or developers to specify which of a Java-9 module's public types are accessible or inaccessible to other modules [177]. A module must explicitly declare which of its public types are accessible to other modules. A module cannot access public types in another module unless those modules explicitly make their public types accessible. As a result, JPMS has added more refined accessibility control—allowing architects and developers to decrease accessibility to packages, reduce the points at which a Java application may be susceptible to security attacks, and design more elegant and logical architectures [96].

Prior to Java 9, the Java platform was a monolith consisting of a massive number of packages, making it challenging to develop, maintain, and evolve. Software developers could not easily choose a subset of the JDK as a platform for their applications. This results in software bloat and more potential points of attack for malicious agents. With the introduction of JPMS in Java 9, the Java platform is now modularized into 95 modules. Furthermore, many internal APIs are hidden from apps using the platform [177], potentially reducing problems involving software bloat and security.

Using JPMS, Java developers can create lightweight custom JREs consisting of only modules they need for their application or the devices they are targeting. As a result, the Java platform can more easily scale down to small devices, which is important for microservices or IoT devices [97]. For example, if a device does not support GUIs, developers could use JPMS to create a runtime environment that does not include the GUI modules, significantly reducing the runtime memory size[96].

Although JPMS allows for specification of prescriptive architectures, the descriptive architecture of a Java application may be inconsistent with the prescriptive architecture. Such inconsistencies may arise due to architects or developers misunderstanding of a software systems' architectures (e.g., an architect mistakenly specifies a more accessible interface than he intended), or simply due to mistaken implementations (e.g., a developer neglects to use a module's interface, even though the architect intended such a use). This can result in (1) a poorly encapsulated architecture, making an application harder to understand and maintain; (2) bloated software; or (3) insecure software. In terms of security, for instance, one of the potential problems is the granting of unnecessary access to internal classes and packages, potentially resulting in security vulnerabilities. In terms of software bloat, inconsistent dependencies can compromise scalability and performance of Java software (e.g., requiring many unnecessary modules from the JDK).

3.2.2 Understanding JPMS Modules

In JPMS, a *module* is a uniquely named, reusable group of related packages, as well as resources (such as images and XML files)[7]. Each module has a descriptor file, `module-info.java`, which contains meta-data, including the declaration of a named module. A named module should specify (1) its dependencies on other modules, i.e., the classes and interfaces that the module needs or expects, and should specify (2) which of its own packages, classes, and interfaces are exposed to other modules.

A module can be a *normal module* or an *open module*. A normal module allows access from other modules at compile time and run time to only explicitly exported packages; an open module allows access from other modules (1) at compile time to only explicitly exported packages and (2) at run time to all its packages [119].

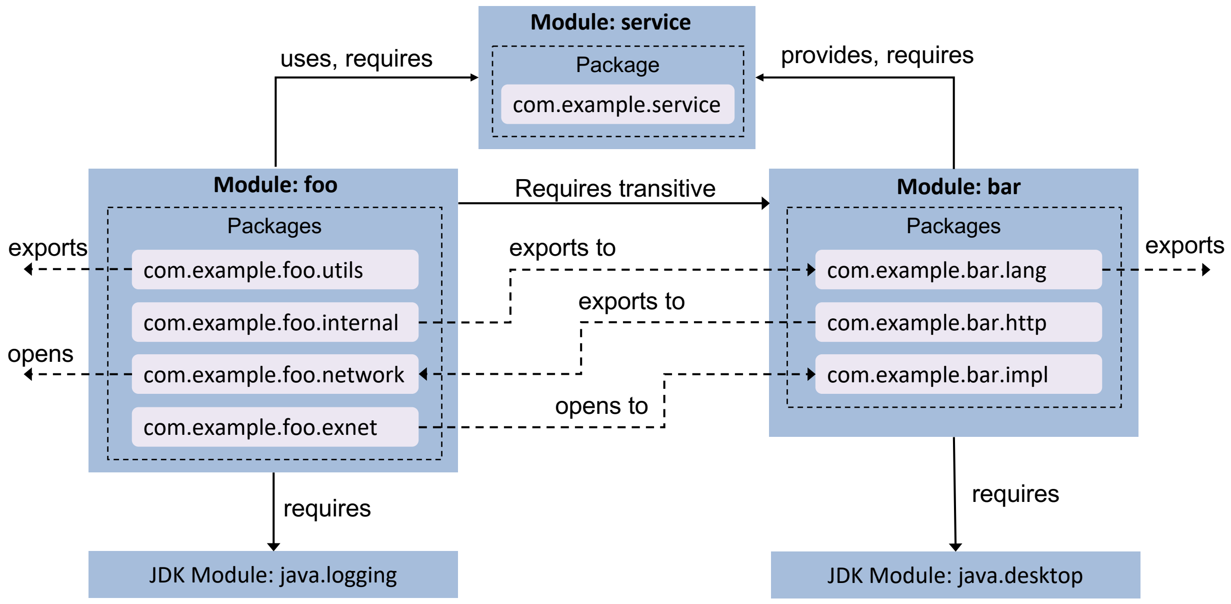
The module declaration file consists of a unique module name and a module body. Any module body can be empty or contain one or more module directives, which specifies a module's exposure to other modules or the modules it needs access to.

Figure 3.1 shows an example of a project with three modules: `bar`, `foo`, and `service`. The declarations of each module provided in its `module-info.java` file is described in Figure 3.1a. Figure 3.1b is a diagram that depicts the relationship between the same modules based on dependencies in their declarations.

A module body can utilize combinations of the following five module directives [119], which specify module interfaces and their usage: the *requires* directive specifies the packages that a module needs

```
module bar {
    requires java.desktop;
    requires service;

    exports com.example.bar.lang;
    exports com.example.bar.http to foo;

    provides com.example.service.Srv with com.example.bar.impl.ImplService;
    }

module foo {
    requires service;
    requires java.logging;
    requires transitive bar;

    exports com.example.foo.utils;
    exports com.example.foo.internal to bar;

    opens com.example.foo.network;
    opens com.example.foo.exnet to bar;

    uses com.example.service.Srv;}

module service {
    exports com.example.service;}

```

(a) Module declarations and their directives provided in their *module-info.java* files.



(b) Specified dependencies between modules based on their directives

Figure 3.1: Three example modules with their inter-dependencies

access to, the *exports* and *opens* directives make packages of a module available to other modules, the *provides* directive specifies the services a module provides, and the *uses* directive specifies the services a package consumes. These directives can be declared as described below:

- The `requires` directive with declaration `requires` m_2 of a module $m1$ specifies the name of a module m_2 that m_1 depends on. m_2 can be a user-defined module or a module within the JDK. For example, in Figure 3.1, module `bar` requires module `java.desktop`. The requires declaration of a module m_1 may be followed by the `transitive` modifier, which ensures that any module m_3 that requires m_1 also implicitly requires module m_2. As an example, in Figure 3.1, module `foo` requires module `bar` and any module that requires `foo` also implicitly requires `bar`.

- The `exports` directive with declaration `exports` p of a module m_1 specifies that m_1 exposes package p's public and protected types, and their nested public and protected types, to all other modules at both runtime and compile time. For example, in Figure 3.1, the module `bar` exports the package `com.example.bar.lang`. We can also export a package specifically to one or more modules by using the `exports` p `to` $m_2, m_3, ..., m_n$ declaration. In this case, the public and protected types of the exported package are only accessible to the modules specified in the *to* clause.

  As an example, in Figure 3.1, module `foo` exports `com.example.foo.internal` to the module `bar`.

- The `opens` directive with declaration `opens` p specifies that package p's nested public and protected types, and the public and protected members of those types, are accessible by other modules at runtime but not compile time. This directive also grants reflective access to all types in p, including the private types, and all its members, from other modules.

  For example, in Figure 3.1, module `foo` makes package `com.example.foo.network` available to other modules only at runtime, including through reflection.

  This directive may also be followed by the *to* modifier, resulting in the `opens` p `to` $m_2, m_3, ..., m_n$

declaration. In this case, the public and protected types of p are only accessible to the modules specified in the *to* clause. For instance, in Figure 3.1, module `foo` uses `opens` directive and makes package `com.example.foo.exnet` available only at runtime, including through reflection, to the module `bar`. Unlike the other directives that can only be used in the body of a module's specification, `open` can be used in both the body of a module's specification and in its header (i.e., before the module's name). The latter usage is a shorthand way of denoting all packages in the module are `open`.

- The `provides with` directive with declaration `provides` c_1 `with` $c_2, c_3, ..., c_n$ of module m_1 specifies that a class c_1 is an abstract class or interface that is provided as a service by m_1. The *with* clause specifies one or more service provider classes for use with `java.util.ServiceLoader`. A service is a well-known set of interfaces and (usually abstract) classes. A service provider is a specific implementation of a service. `java.util.ServiceLoader<S>` is a simple service-provider loading facility. It loads a provider implementing the service type `S` [2]. For instance in Figure 3.1, module `bar` provides the abstract class `com.example.service.Srv` as a service using the `com.example.bar.impl.ImplService` class as the service's implementation.

- The `uses` directive with declaration `uses` c_1 of a module m_1 specifies that m_1 uses a service object of an abstract class or interface, c_1, provided by another module. For this purpose, the module should discover providers of the specified service via `java.util.ServiceLoader`. As an example from Figure 3.1, module `foo` uses the service object of class `com.example.service.Srv`, which is provided by module `bar`.

Note that, as depicted in Figure 3.1, both `provides with` and `uses` directives need the module being declared to `require` the service module as well.

3.3 Inconsistent Module Dependencies

Based on the module directives described in Section 3.2.2, inconsistencies may arise when using modules. Insufficiently specified dependencies (e.g., a module that attempts to use a package it does not have a requires directive for) are already checked by the Java platform. However, *excess dependencies*, where a module either (1) exposes more of its internals than are used or (2) requires internals of other modules that it never uses, are not handled by Java. These inconsistencies can affect various architectural attributes:

A1: Encapsulation and Maintenance—Requiring unneeded functionalities of other modules increases the complexity of the module unnecessarily, compromises its encapsulation, and decreases its maintainability.

A2: Software Bloat and Scalability—Requiring unneeded modules, especially from JDK, can result in bloated software, which compromises scalability of the application.

A3: Security—Excessively exposing the internals of a module can result in errors or security issues arising in the module.

To achieve a systematic and comprehensive coverage of all types of inconsistent module dependencies, we studied all potential inconsistencies resulting from developers' misuse of each type of module directive. In the remainder of this section, we focus on specifying eight types of inconsistent dependencies that may arise when using JPMS and the functions needed to specify those dependencies.

Table 3.1 includes 11 functions that directly model different variations of the five module directives in JPMS. To describe a class loading a service using `java.util.ServiceLoader` API, we define the *LoadsService* function. For actual code usage among packages, as opposed to those specified through module directives, we define the *Dep* function.

Table 3.1: Functions describing dependencies based on module directives of JPMS

Function	Description
$Req(m_1, m_2)$	Module m_1 requires module m_2.
$ReqJDK(m_1, m_{jdk})$	Module m_1 requires the JDK module m_{jdk}.
$ReqTransitive(m_1, m_2)$	Module m_1 requires transitive module m_2.
$Exp(m, p)$	Module m exports package p.
$ExpTo(m_1, p_1, \{m_2, m_3, \ldots\})$	Module m_1 exports package p_1 to the set of modules $\{m_2, m_3, \ldots\}$.
$Open(m)$	Module m is open.
$Opens(m, p)$	Module m opens package p.
$OpensTo(m_1, p, \{m_2, m_3, \ldots\})$	Module m_1 opens package p to the set of modules $\{m_2, m_3, \ldots\}$.
$Uses(m, s)$	Module m uses Service s.
$ProvidesWith(m, s, \{c_1, c_2, \ldots\})$	Module m provides service s with the set of classes $\{c_1, c_2, \ldots\}$.
$LoadsService(c, s)$	Class c loads Service s via the `java.util.ServiceLoader` API.
$Dep(p_1, p_2)$	Source code in package p_1 uses classes of package p_2.
$ReflDep(p_1, p_2)$	Source codes in package p_1 uses classes of package p_2 via reflection.

By leveraging the functions in Table 3.1, we introduce eight types of excess inconsistent dependencies: requires, JDK requires, requires transitive, exports(to), provides with, uses, open, and opens(to) modifiers. For each inconsistent dependency type, there is a dependency explicitly defined in a `module-info` file which is not actually used in the source code of the module. Using these formal definitions, Section 3.4 detects and repairs the following inconsistent dependencies.

Inconsistent Requires Dependency: This scenario describes an inconsistent *requires* dependency in which (1) module m_1 explicitly declares that it requires another module m_2 and (2) no class of m_1 actually uses any class inside exported packages of m_2. As a result, this inconsistency mostly affects attribute A1. It can also affect attribute A2.

$$Req(m_1, m_2) \land (\nexists\, p_1 \in m_1,\ p_2 \in m_2 : Dep\ (p_1, p_2)) \tag{3.1}$$

Inconsistent JDK Requires Dependency: This scenario describes an inconsistent *requires* dependency in which module m_1 explicitly declares that it requires a module inside the Java JDK, m_{jdk}. However, none of the classes inside m_1 uses any class inside exported packages of m_{jdk}. Hence, it affects attribute A1, and more importantly A2. We distinguish this scenario from the previous one because an inconsistency involving JDK modules has a greater effect on portability than the previous more generic scenario.

$$Req(m_1, m_{jdk}) \wedge (\nexists p_1 \in m_1, p_2 \in m_{jdk} : Dep(p_1, p_2)) \tag{3.2}$$

Inconsistent Requires Transitive Dependency: An excess *transitive* modifier in a *requires* dependency consists of the following (1) a module m_1 explicitly declares in its `module-info` file that it transitively requires another module m_2—which means any module that requires m_1 also implicitly requires m_2; and (2) no class of a module that requires $m1$ actually uses any class in $m2$. This type of inconsistency mostly affects attribute A1, but also affects A2.

$$\begin{aligned}&ReqTransitive(m_1, m_2) \wedge (\forall\, m : Req(m, m_1),\\&\forall\, p \in m, \forall\, p_2 \in m_2 : \neg Dep(p, p_2))\end{aligned} \tag{3.3}$$

Inconsistent Exports/Exports to Dependency: An inconsistent *exports* dependency occurs when a module m_1 explicitly exports a package p_1 to all other modules, while no package in those other modules use p_1.

$$Exp(m_1, p_1) \wedge (\forall\, p \notin m_1 : \neg Dep(p, p_1)) \tag{3.4}$$

For an *exports to* directive, this inconsistency occurs when m_1 exports the package p_1 to a specific list of modules M, while no class outside m_1, or inside module list M, uses any class inside p_1.

$$ExpTo(m_1, p_1, M) \wedge (\forall\, p \in M : \neg Dep(p, p_1)) \tag{3.5}$$

These inconsistencies mostly affect attribute A3 by granting unnecessary access to classes and packages. They also affect attribute A1 due to complicating the architecture.

Inconsistent Provides With Dependency: An inconsistent *provides with* dependency has two key parts: (1) a module *m* explicitly declares that it provides a service *s*, which is an abstract class or interface that is extended or implemented by a set of classes $E = \{c_1, c_2, ..., c_k\}$ inside *m*; and (2) none of the classes inside other modules uses service *s* via the `java.util.ServiceLoader` API. Consequently, this inconsistency type—similar to inconsistent `requires` dependency—affects attribute A1 and A2 because the `provides with` dependency necessitates a `requires` directive as well. Additionally, this inconsistency type grants unnecessary access to a subset of the application's classes via the `ServiceLoader` API which affects attribute A3.

$$ProvidesWith(m,s,E) \land (\forall m' \neq m : \neg Uses(m',s)) \tag{3.6}$$

Inconsistent Uses Dependency: An inconsistent *uses* dependency occurs when (1) a module *m* explicitly declares in its `module-info.java` file that it uses a service *s* and (2) none of the classes inside *m* actually use the service *s* via the `java.util.ServiceLoader` API. This inconsistency type, similar to the previous type, will affect attribute A1 and A2, due to adding an additional `requires` directive.

$$Uses(m,s) \land (\forall\ c \in m : \neg\ LoadsService(c,s)) \tag{3.7}$$

Inconsistent Open Modifier: An excess *open* modifier occurs in the following scenario: (1) a module *m* declares that it opens all its packages to all other modules—recall from Section 3.2.2 that unlike the other directives, `open` can be used in the header of a module's specification to denote all its packages are open; and (2) there is at least one package *p* inside *m* that no class outside *m* reflectively accesses. As a result, any such package *p* is potentially open to misuse through reflection, e.g., external access to private members of a class that should not be allowed by any other class. This inconsistency type will affect attribute A3—and make the architecture inaccurate and more complicated, affecting attribute A1.

$$Open(m) \wedge (\exists p \in m : \forall p' \notin m : \neg ReflDep(p', p) \tag{3.8}$$

Inconsistent Opens/Opens To: An inconsistent *opens* dependency occurs when a module m declares that it opens a package p to all other modules via reflection, while none of the classes outside m reflectively accesses any classes of package p.

$$Opens\ (m, p) \wedge \forall\ p' \notin m : \neg ReflDep(p', p) \tag{3.9}$$

Similarly, for *opens to*, the *to* modifier specifies a list of modules M for which module m opens a package p to access via reflection, while no package of m reflectively accesses p.

$$OpensTo(m, p, M) \wedge \forall\ p' \in M : \neg ReflDep(p', p) \tag{3.10}$$

For these inconsistency types, private members of p are open to dangerous misuse through undesired access and reflection, affecting attribute A3, and can also affect attribute A1 due to unnecessarily complicating the architecture.

3.4 DARCY: Framework Overview

In the previous section, we introduced various types of inconsistent dependencies. This section describes how we leverage these definitions to design and implement DARCY. Figure 3.2 depicts a high-level overview of DARCY comprised of two phases, *Detection* and *Repair*. DARCY is implemented in Java and Python.

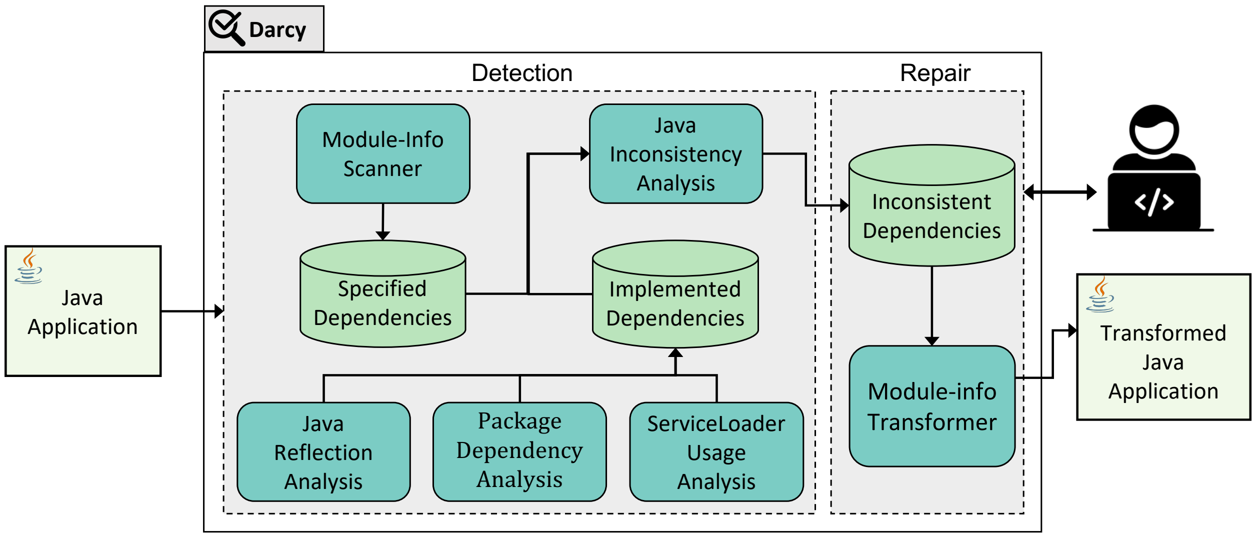


Figure 3.2: A high-level overview of DARCY

3.4.1 Detection

The detection phase takes a Java application as input and identifies any instance of the eight inconsistent dependencies described in Section 3.3.

To identify the implemented dependencies of an input Java application, DARCY relies on static analysis, represented as *Package Dependency Analysis* in Figure 3.2. In the implementation of DARCY, we leveraged Classycle [106] for *Package Dependency Analysis*. More precisely, the information about the implemented dependencies in the source code of the input application is collected by running Classycle, which provides a complete report of all dependencies in source code of a Java application at both the class and package levels. we only need the extracted dependencies among packages since the dependencies defined in modules are at the package level. *Package Dependency Analysis's* results are stored in *Implemented Dependencies*, which is a database component.

A Java application may contain multiple modules, each with a `module-info` file describing the module's dependencies. For extracting a prescriptive architecture, we developed *Module-Info Scanner* which examines all `module-info.java` files within the input Java application and extracts all specified dependencies which are defined at the package level. The collected information of specified dependencies are stored in another database component, *Specified Dependencies*.

Java Reflection Analysis leverages a custom static analysis [113], which we have implemented using the Soot framework [200], to identify usage of reflection in the input application. The traces of any actual usage of reflection in the Java application is then stored in *Implemented Dependencies*.

Java Reflection Analysis extracts reflective invocations that occur in cases where non-constant strings, or inputs, are used as target methods of a reflective call. Reflective invocation of a method, for both constructor and non-constructor methods, occurs in three stages: (1) class procurement (i.e., a class with the method of interest is obtained) (2) method procurement (i.e., the method of interest to be invoked is identified), and (3) the method of interest is actually invoked. *Java Reflection Analysis* attempts to identify information at each stage.

```
1  ClassLoader cl = MyClass.getClassLoader();
2  try { Class c = cl.loadClass("NetClass");
3        ...
4        Method m = c.getMethod("getAddress",...);
5        ...
6        m.invoke(...); }
7  catch { ... }
```

Figure 3.3: Reflective method invocation example

A simple example, based on those found in real-world apps, of reflective method invocation, not involving constructors, is depicted in Figure 3.3. In this example, a **`ClassLoader`** for **`MyClass`** is obtained (line 1), which is responsible for loading classes. The **`NetClass`** class is loaded using that **`ClassLoader`** (line 2). The **`getAddress`** method of **`NetClass`** (line 4)—which performs network operations—is retrieved and eventually invoked using reflection (line 6).

Our analysis identifies reflectively invoked methods using a backwards analysis. That analysis begins by identifying all reflective invocations (e.g., line 6 in Figure 3.3). Next, the analysis follows the use-def chain of the invoked **`java.lang.reflect.Method`** instance (e.g., **`m`** on line 6) to identify all possible definitions of the **`Method`** instance (e.g., line 4). Our analysis considers various methods that return **`Method`** instances, i.e., using **`getMethod`** or **`getDeclaredMethod`** of **`java.lang.Class`**.

The analysis then records each identified method name. If the analysis cannot resolve the name, this information is also recorded. In this case, the analysis conservatively indicates that any method of the package opened for reflection can be accessed.

For constant strings, the analysis attempts to identify the class name that is being invoked. Similar to the resolution of method names, the analysis follows the use-def chain of the **`java.lang.Class`** instance from which a **`java.lang.Class`** is retrieved (e.g., following the use-def chain of **`c`** on line 4). we model various means of obtaining a **`java.lang.Class`** instance. For example, the class may be loaded by name using a **`ClassLoader`**'s **`loadClass(...)`** method (e.g., line 2), using **`java.lang.Class`**'s **`forName`** method, or through a class constant (e.g., using **`NetClass.class`**). The analysis then records the class name it can find statically, or stores that it could not resolve that name. Note that our analysis considers any subclass of **`ClassLoader`**. Our reflection analysis involving constructors works in a similar manner by analyzing invocations of **`java.lang.reflect.Constructor`** and invocations of its **`newInstance`** method.

Similar to our analyses for reflectively invoked methods, we perform analyses for any **`set*`** methods of **`java.lang.reflect.Field`** (e.g., **`setInt(...)`**) or **`get*Field*`** methods of **`java.lang.Class`** (e.g., **`getDeclaredField(String)`**).

For extracting the implemented dependencies of type *uses* we implemented *ServiceLoader Usage Analysis* which leverages a custom static analysis using the Soot framework to identify usage of **`java.util.ServiceLoader`** in the input application. The traces of any actual usage of a service is then stored in *Implemented Dependencies*.

An application obtains a service loader for a given service by invoking the static **`load`** method of **`ServiceLoader`** API. A service loader can locate and instantiate providers of the given service using the **`iterator`** or **`stream`** method [18], through which an instance of each of the located service providers can be created. As an example, Figure 3.4 depicts the code that obtains a **`ServiceLoader`** for **`MyService`** (line 1). The **`ServiceLoader`** loads providers of **`MyService`** (line 2) and can instantiate any of the located providers of this service using its iterator—created by the for loop in line 3. In this example, the service provider with the **`getService`** method is desired (line 4).

```
1    ServiceLoader<MyService> loader;
2    loader = ServiceLoader.load(MyService.class);
3    for (MyService s : loader) {
4     if (s.getService != null){... } }
```

Figure 3.4: Service loader example

Our analysis identifies the usage of the **`ServiceLoader`** API using a backward analysis by following the use-def chain of **`ServiceLoader`** instances (e.g., **`s`** on line 4) to identify all possible definitions of a **`ServiceLoader`** (e.g, line 2 in Figure 3.4). The results of the **`ServiceLoader`** API usage is then stored in *Implemented Dependencies*.

Java Inconsistency Analysis's main goal is to identify all types of inconsistency scenarios described in Section 3.3. For each directive in a `module-info.java` file, *Java Inconsistency Analysis* explores implemented and specified dependencies, stored in their respective database components, to identify any occurrence of an inconsistent dependency defined in Section 3.3. If a matching instance is found, *Java Inconsistency Analysis* reports the identified architectural inconsistency, the module affected, and the specific directive involved. The component then stores the identified inconsistencies in *Inconsistent Dependencies*, which are then used in the repair phase.

3.4.2 Repair

To repair inconsistent dependencies, *Module-Info Transformer* deletes or modifies the explicit dependencies defined in the `module-info` files. Inconsistencies found in the previous phase are all unnecessarily defined dependencies among an application's modules and packages. Therefore, *Module-Info Transformer* needs to omit those inconsistent dependencies specified in the `module-info` files.

The result of the detection phase includes the type and details of identified inconsistencies. For instance, in the case of an inconsistent *exports* dependency, one result stored in *Inconsistent Dependencies* includes the module in which this dependency is specified, the type of the inconsistent dependency (*exports* in this case), and the package that is unnecessarily exported. The repair phase takes the results of the detection phase as input. For each module, the repair phase finds the related records of inconsistent dependencies defined in that module and modifies the affected lines in `module-info`.

For this purpose, we leveraged ANTLR [10] to transform the `module-info.java` files to repair the inconsistent dependencies. ANTLR is a parser generator for reading, processing, executing, or translating a structured text. Hence, we generated a customized parser using Java-9 grammar so that we can modify it to check the records of inconsistent dependencies found in the detection phase of DARCY.

More precisely, we have implemented the generated parser so that, if it finds any match between the tokens of `module-info` files and the inconsistent dependencies, it skips or modifies the specific token with respect to the type of the inconsistency. As a result, depending on the type of dependency, the corresponding line in the `module-info` file is omitted or modified.

Module-Info Transformer repairs each type of inconsistent dependency. In most cases, *Module-Info Transformer* deletes the entire statement. However, for `requires transitive`, *Module-Info Transformer* only removes the token `transitive`.

In case of inconsistencies involving `open` module *m* (Equation 3.8 in Section 3.3), the `open` modifier is removed from the header of the module declaration. However, there may be some packages in *m* that other modules reflectively access. For each of these packages, *Module-Info Transformer* adds an `opens to` statement that make private members of the package accessible to the modules that reflectively access the package. If there is no package in *m* that is reflectively accessed by other modules, no statement will be added to the module's body.

In certain situations, the DARCY user may disagree with the way it repairs and modifies the specified dependencies because DARCY is not aware of the architect's or developer's intentions. For example, this situation may occur if the user wants to develop a library and export some packages for further needs or even allow other modules to reflectively access the internals of some classes and packages. DARCY warns the developers and architects about potential threats caused by architectural inconsistencies in their Java application, and allows them to override DARCY prior to application of repairs.

3.5 Evaluation

To assess the effectiveness of DARCY, we study the following research questions:

RQ1: How pervasive are inconsistent, architectural dependencies in practice?
RQ2: How accurate is DARCY at detecting inconsistent, architectural dependencies and repairing them at build-time?
RQ3: To what extent does DARCY reduce the attack surface of Java modules?
RQ4: To what extent does DARCY enhance encapsulation of Java modules?
RQ5: To what extent does DARCY reduce the size of runtime memory?
RQ6: What is DARCY's runtime efficiency in terms of execution time?

To answer these research questions, we selected a set of Java applications from GitHub [11], a large and widely used open-source repository of software projects, all of which are implemented in Java 9+. For this purpose, we crawled GitHub using a Python script that leverages Beautiful Soup [1], i.e., a web scraping library and GitHub's API. The script assigns a query to the original GitHub URL, where it increments pages and filters Java applications by containing a file named

Table 3.2: Subject applications

No.	Application Name	# Modules	# Packages	# Directives
1	Auto-sort	3	35	13
2	Ballerina-lang	65	4486	532
3	Blynk-Server	9	742	122
4	BunnyHop	2	22	28
5	Codersonbeer-app	4	5	9
6	Constantin	4	15	9
7	Eclipse Jetty	56	10437	413
8	Java-9-lab	5	6	15
9	Java-9-modularity	4	5	11
10	Java-Bookstore	6	11	17
11	Java-SPI	6	6	26
12	Java9-demo	4	12	10
13	Java9-junit	3	15	13
14	Java9-labs	4	6	10
15	java9-modules	2	2	5
16	Java9-TLB-modules	5	8	12
17	JavaUtils	6	39	36
18	Jigsaw-resources	2	2	5
19	Jigsaw-tst	4	5	11
20	Jwtgen	2	9	13
21	Logback	2	1250	65
22	Meetup	4	4	14
23	Music-UI	3	7	15
24	Number-to-text	3	11	11
25	Practical-Security	4	8	20
26	Quasar	4	783	42
27	QuestDB	2	3222	65
28	Rahmnathan-utils	3	16	14
29	RecolInline-Server	7	73	42
30	Rhizomatic-IO	7	111	58
31	Sense-nine	6	48	31
32	Sirius	7	392	37
33	Spring-mvn-java9	3	6	18
34	Springuni-java9	3	2	6
35	Tascalate-Javaflow	10	258	56
36	The-Message	3	32	16
37	TRPZ	4	20	19
38	Vstreamer	6	6	25

`module-info.java`. Subsequently, we filtered GitHub repositories with more than one module that could have been successfully built. Our search covered thousands of GitHub repositories, and our final evaluation dataset resulted in 52 Java 9+ applications, avoiding any selection bias toward our approach. DARCY detected 567 instances of architectural inconsistencies in 38 applications. Table 3.2 includes these 38 subject applications, their number of modules, packages, and directives.

3.5.1 RQ1: Pervasiveness

Table 3.3 shows, for each application in our dataset, the total number of inconsistent dependencies DARCY found and separates them by their type. 73% of applications in our dataset (38 out of 52) have a total number of 567 inconsistent dependencies. Recall that even one existing inconsistent dependency could cause undesired behaviors or issues with encapsulation, security, or memory utilization (see Section 3.3.)

Table 3.3: Identified inconsistencies and robustness results

Application	# Total	Inconsistencies Types							% Correct	Compiled	Test Passing Rate (%)	
Name	Incons.	R	R.J.	R.T	E	O	P	U	Incons.	(After Repair)	Before	After
Auto-sort	1	-	-	-	1	-	-	-	100	✓	97	97
Ballerina-lang	115	21	4	-	88	1	1	-	100	✓	100	100
Blynk-Server	26	-	-	-	26	-	-	-	100	✓	100	100
BunnyHop	17	-	1	2	11	3	-	-	100	✓	-	-
Codersonbeer-app	4	1	-	-	2	1	-	-	100	✓	-	-
Constantin	5	-	-	1	4	-	-	-	100	✓	100	100
Eclipse Jetty	147	3	1	90	49	-	4	-	100	✓	100	100
Java-9-lab	1	-	-	-	1	-	-	-	100	✓	-	-
Java-9-modularity	1	-	-	1	-	-	-	-	100	✓	-	-
Java-Bookstore	3	-	-	2	-	-	1	-	100	✓	-	-
Java-SPI	5	-	-	1	4	-	-	-	100	✓	-	-
Java9-demo	1	-	-	-	1	-	-	-	100	✓	-	-
Java9-junit	1	-	-	-	1	-	-	-	100	✓	-	-
Java9-labs	4	-	-	-	4	-	-	-	100	✓	-	-
java9-modules	1	-	-	-	1	-	-	-	100	✓	-	-
Java9-TLB-modules	1	-	-	-	1	-	-	-	100	✓	-	-
JavaUtils	29		14	9	6	-	-	-	100	✓	-	-
Jigsaw-resources	1	-	-	-	1	-	-	-	100	✓	-	-
Jigsaw-tst	1	-	-	-	1	-	-	-	100	✓	-	-
Jwtgen	2	1	-	-	1	-	-	-	100	✓	-	-
Logback	30	-	-	3	26	-	1	-	100	✓	100	100
Meetup	6	-	-	-	3	-	3	-	100	✓	-	-
Music-UI	4	-	-	1	1	2	-	-	100	✓	-	-
Number-to-text	4	-	-	-	4	-	-	-	100	✓	100	100
Practical-Security	4	-	1	-	3	-	-	-	100	✓	-	-
Quasar	22	-	1	4	17	-	-	-	100	✓	100	100
QuestDB	34	-	-	1	33	-	-	-	100	✓	100	100
Rahmnathan-utils	7	-	1	-	6	-	-	-	100	✓	-	-
RecolInline-Server	20	4	6	2	6	2	-	-	100	✓	-	-
Rhizomatic-IO	3	-	-	-	2	-	-	1	100	✓	100	100
Sense-nine	10	2	2		3	3	-	-	100	✓	-	-
Sirius	9	2	-	4	3	-	-	-	100	✓	-	-
Spring-mvn-java9	8	2	-	-	6	-	-	-	100	✓	-	-
Springuni-java9	3	2	-	-	1	-	-	-	100	✓	-	-
Tascalate-Javaflow	17	-	-	12	5	-	-	-	100	✓	-	-
The-Message	9	-	-	-	9	-	-	-	100	✓	-	-
TRPZ	5	-	-	-	3	2	-	-	100	✓	-	-
Vstreamer	6	1	-	-	2	-	-	3	100	✓	-	-
Total	567	39	31	133	336	14	10	4				

As depicted in Table 3.3, most of the inconsistent dependencies are of types *exports* or *requires* because these two types of directives are used more frequently than others. The high frequency of inconsistent *exports* dependencies indicates that granting unnecessary access to internal packages are quite common in Java 9+ applications, which could cause security vulnerabilities. Furthermore, different types of inconsistent *requires* dependencies impair the applications' encapsulation and comprise their maintainability. Additionally, the *requires JDK* inconsistency increases the risk of loading unnecessary JDK modules and compromising portability.

Table 3.3 indicates that a few applications have inconsistent dependencies of type *provides with*, and only two applications have an inconsistent *uses* dependency. In fact, these directives are rare compared to other directives. For *provides with* and *uses*, Java checks most of the requirements for avoiding inconsistent dependencies at build-time. Therefore, the possibility of defining an inconsistent *provides with* and *uses* dependencies decreases. Nevertheless, DARCY covers the inconsistent dependencies corresponding to these two directives because they are risky and may appear more frequently in future usage of JPMS.

3.5.2 RQ2: Correctness

To answer RQ2 for DARCY's detection capability at build-time, we ran the detection phase for each Java application in our evaluation dataset before running them and assessed whether DARCY can accurately detect inconsistent dependencies. To that end, we manually checked the inconsistent dependencies found by DARCY to ensure their correctness. More precisely, we compared the corresponding record in both *Implemented Dependencies* and *Specified Dependencies* to verify the correctness of the inconsistencies discovered by the detection phase. As described in Table 3.3, the result shows that all inconsistent dependencies found by DARCY at build-time are correct.

To evaluate DARCY's ability to correctly resolve inconsistencies at build-time, we ran the repair phase of DARCY for each Java application in our evaluation dataset before running them and assessed whether DARCY repairs the detected inconsistencies without introducing any unexpected behavior. To assess the correctness of a repair, we (1) check if each application compiles successfully after running the repair phase and (2) if the application contains a test suite, determine if the application obtains the same *test passing rate*, i.e., the ratio of the number of passing test cases to the total number of test cases, both before and after repairs. We also ran the detection phase after the repair actions. The result showed zero inconsistencies within the transformed Java applications.

The results for compilation after the repair phase are shown in Table 3.3, indicating that all transformed Java applications compiled successfully. This confirms that the inconsistent dependencies have been robustly resolved in a way that does not prevent compilation of the applications. Additionally, eleven applications in our study contain a test suite. The passing rate for each of these test suites remains the same both before and after DARCY repairs, demonstrating that DARCY does not negatively affect the expected behavior of repaired applications.

3.5.3 RQ3: Security

To assess DARCY's ability to enhance security, we consider the *attack surface* of Java 9+ applications. The attack surface of a system is the collection of points at which the system's resources are externally visible or accessible to users or external agents. Manadhata et al. introduced an attack-surface metric to measure the security of a system in a systematic manner [155, 156, 157]. Every externally accessible system resource can potentially be part of an attack and, hence, contributes to a system's attack surface. This contribution reflects the likelihood of each resource being used in security attacks. Intuitively, the more actions available to a user or the more resources that are accessible through these actions, the more exposed an application is to security attacks [155, 156, 157].

For a Java 9+ application, the main resource under consideration is a Java module. As a result, we define the attack surface of an application as the number of packages that are accessible from

Table 3.4: Result for attack-surface reduction at build-time

Application Name	# exposed pckg (before)	# exposed pckg (after)	Attack Surface Reduction (%)
Auto-sort	2	1	50.00%
Ballerina-lang	195	106	45.64%
Blynk-Server	58	32	44.83%
BunnyHop	16	2	87.50%
Codersonbeer-app	4	1	75.00%
Constantin	4	0	100.00%
Eclipse Jetty	117	68	41.88%
Java-9-lab	3	2	33.33%
Java-9-modularity	2	2	0.00%
Java-Bookstore	4	4	0.00%
Java-SPI	6	3	50.00%
Java9-demo	2	1	50.00%
Java9-junit	4	3	25.00%
Java9-labs	5	1	80.00%
java9-modules	2	1	50.00%
Java9-TLB-modules	5	4	20.00%
JavaUtils	7	1	85.71%
Jigsaw-resources	2	1	50.00%
Jigsaw-tst	3	2	33.33%
Jwtgen	1	0	100.00%
Logback	50	24	52.00%
Meetup	4	1	75.00%
Music-UI	5	2	60.00%
Number-to-text	4	3	25.00%
Practical-Security	4	3	25.00%
Quasar	22	5	77.27%
QuestDB	52	19	63.46%
Rahmnathan-utils	6	0	100.00%
RecolInline-Server	17	9	47.06%
Rhizomatic-IO	13	11	15.38%
Sense-nine	6	1	83.33%
Sirius	14	11	21.43%
Spring-mvn-java9	6	0	100.00%
Springuni-java9	1	0	100.00%
Tascalate-Javaflow	11	6	45.45%
The-Message	10	9	10.00%
TRPZ	5	3	40.00%
Vstreamer	3	1	66.67%
Avg. Attack Surface Reduction			56.37%

outside its modules. To measure the attack surface of Java 9+ applications, we count the number of packages exposed by *exports (to)* and *open(s to)* directives. These directives make internals of packages accessible to other modules.

As shown in Table 3.4, 36 out of 38 applications had an average attack-surface reduction of about 56% at their build-time. DARCY was able to totally eliminate the attack surface in 5 applications.[2] Although eliminating the module-based attack surface does not result in perfect security, DARCY can maximize protection to the asset (i.e., Java packages) through a module's interfaces by eliminating all unnecessary exports and opens directives of the module—other attack vectors (e.g., IPC over network sockets) still remain but are out of scope for DARCY.

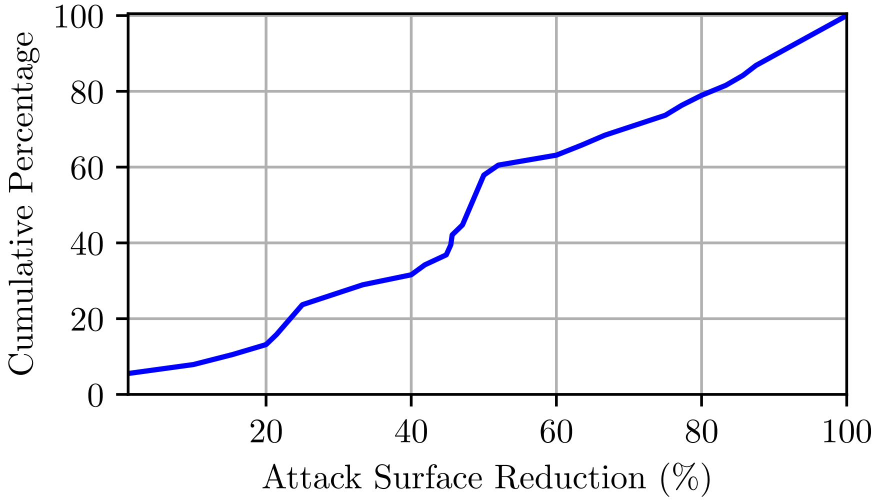


Figure 3.5: Cumulative distribution function (CDF) of the attack surface reduction at build-time

Figure 3.5 shows a cumulative distribution function (CDF) of the ratio of attack surface reduction by DARCY among the applications in our dataset at build-time. It indicates that in 60% of the applications, DARCY was able to reduce the applications' attack surface up to about 50% at build-time. Furthermore, in 80% of the applications, the repairs by DARCY reduced the applications' attack surface up to 80%, which means that for the remaining 20% of the applications, it reduced the applications' attack surface by even more than 80% at their build-time.

The relatively large reduction of the attack surface in applications achieved by DARCY indicates that it can significantly curtail security risks in Java 9+ applications.

[2]These applications are essentially software utilities or libraries including different modules that provide functionalities for different situations, but do not have any dependency on one another.

3.5.4 RQ4: Encapsulation

To evaluate the ability of DARCY to enhance the encapsulation of Java 9+ applications, we leveraged two metrics selected from an extensive investigation by Bouwers et al. [72] about the quantification of encapsulation for implemented software architectures. we selected metrics that involve architectural dependencies and are appropriate for the context of modules in JPMS and Java 9+ applications.

The first metric we selected is Ratio of Coupling (RoC) [74], which measures coupling among an application's modules. For Java 9+ modules, RoC is the ratio of the *number of existing dependencies among modules* to the *number of all possible dependencies among modules*. Ideally, the value of RoC would be low, meaning that only a small part of all possible dependencies among modules is actually utilized—making it less likely that faults, failures, or errors introduced by changes or additions to modules will propagate across modules.

The second metric we selected is a variant of Cumulative Component Dependency (CCD) [143] which is the sum of all outgoing dependencies for a component. For Java 9+ modules, outgoing dependencies are *requires* and *uses* dependencies of each module. The specific variant we used is Normalized CCD (NCD), which is the ratio of CCD for each module to the total number of modules. Ideally, the value of CCD, or NCD, is low, indicating lower coupling and better encapsulation.

Table 3.5 presents the amount of RoC and NCD change in 38 Java 9+ applications with inconsistent dependencies at build-time. Across all 38 applications, the amount of RoC is reduced by an average of about 28%, and up to about 81% at build-time. The amount of NCD is also reduced in 24 applications by an average of about 21%, and up to 79% at build-time.

Figure 3.6 shows the cumulative distribution function (CDF) of the RoC reduction followed by the DARCY's repairs at build-time. It indicates that, ay build-time, in about 70% of applications, DARCY was able to reduce the RoC up to 40%.

Table 3.5: Results for encapsulation improvement at build-time

Application Name	# Directives (before)	RoC % Change	NCD % Change
Auto-sort	13	7.69%	-
Ballerina-lang	532	21.43%	7.46%
Blynk-Server	122	21.31%	-
BunnyHop	28	60.71%	25.00%
Codersonbeer-app	13	30.77%	12.50%
Constantin	9	55.56%	20.00%
Eclipse Jetty	413	35.59%	34.18%
Java-9-lab	15	6.67%	-
Java-9-modularity	11	9.09%	12.50%
Java-Bookstore	17	17.65%	16.67%
Java-SPI	26	15.38%	5.56%
Java9-demo	10	10.00%	-
Java9-junit	13	7.69%	-
Java9-labs	10	40.00%	-
java9-modules	5	20.00%	-
Java9-TLB-modules	12	8.33%	-
JavaUtils	36	80.56%	79.31%
Jigsaw-resources	5	20.00%	-
Jigsaw-tst	11	9.09%	-
Jwtgen	13	15.38%	8.33%
Logback	65	46.15%	21.43%
Meetup	14	42.86%	-
Music-UI	15	26.67%	10.00%
Number-to-text	11	9.09%	-
Practical-Security	20	10.00%	7.69%
Quasar	42	52.38%	25.00%
QuestDB	65	52.31%	8.33%
Rahmnathan-utils	14	50.00%	12.50%
RecollInline-Server	42	47.62%	48.00%
Rhizomatic-IO	58	5.17%	2.44%
Sense-nine	31	29.03%	16.00%
Sirius	37	24.32%	26.09%
Spring-mvn-java9	18	44.44%	16.67%
Springuni-java9	6	50.00%	40.00%
Tascalate-Javaflow	56	30.36%	26.67%
The-Message	16	6.25%	-
TRPZ	19	10.53%	-
Vstreamer	25	16.00%	20.00%

Total # of Affected Systems (RoC)	28
RoC Reduction Avg.	25.34%
Total # of Affected Systems (NCD)	15
NCD Reduction Avg.	20.73%

Figure 3.7 demonstrates the cumulative distribution function (CDF) of DARCY's NCD reduction rate at build-time. It shows that resolving the inconsistencies in 80% of the applications could

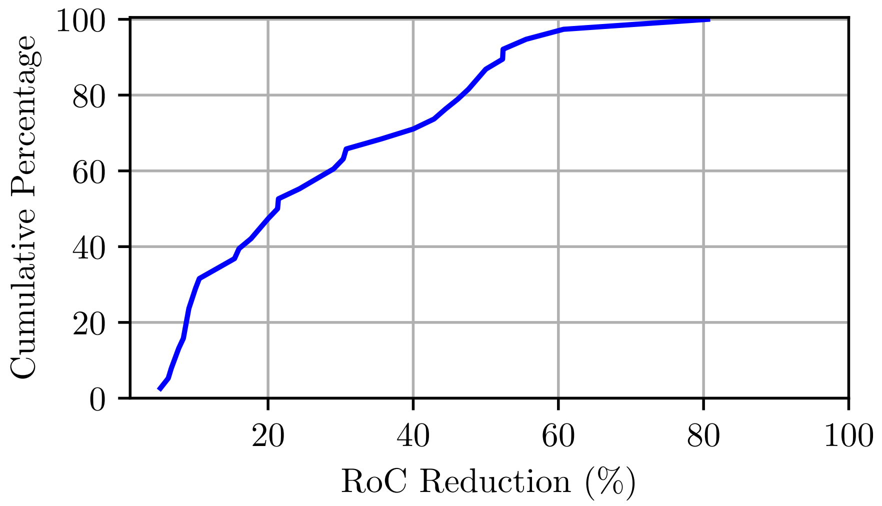


Figure 3.6: Cumulative distribution function (CDF) of the Ratio of Coupling (RoC) reduction at build-time

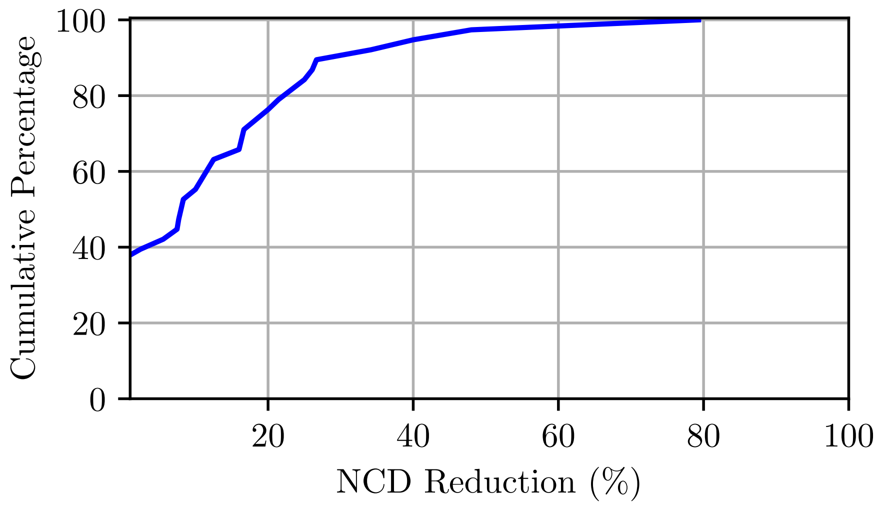


Figure 3.7: Cumulative distribution function (CDF) of the Normalized Cumulative Component Dependency (NCD) reduction at build-time

reduce the NCD up to 22%.

These results indicate that DARCY can successfully enhance the encapsulation of Java 9+ applications by a significant amount.

3.5.5 RQ5: Software Bloat

To answer this research question, we measured the runtime memory needed by each application before and after DARCY's repair phase. Recall the fact that in Java 9+, with the JDK being modularized, we are able to create a lightweight custom Java Runtime Environment (JRE), reducing software bloat. More specifically, the size of a custom JRE may be reduced after a repair if the application has inconsistent dependencies of type *requires JDK* (Equation 3.2 of Section 3.3).

Table 3.6: Results for software-bloat reduction

Application Name	Runtime Memory Reduction (%)
Ballerina-lang	8.29%
BunnyHop	54.72%
Eclipse Jetty	16.83%
Java-SPI	6.04%
JavaUtils	21.87%
Practical-Security	0.76%
Quasar	4.55%
Rahmnathan-utils	0.12%
RecolInline-Server	50.70%
Sense-nine	0.63%
Avg. Memory Reduction	16.45%

Table 3.6 shows the reduction of software bloat in terms of the reduction in the size of the JRE image of affected applications after removing inconsistent *requires JDK* dependencies at build-time. According to the results, the reduction is about 16% in 10 applications and up to 55%.

Figure 3.8 demonstrates the cumulative distribution function (CDF) of DARCY's software bloat reduction rate at build-time. The figure shows that in 80% of the applications DARCY could reduce the size of the JRE image up to about 22% by resolving the module inconsistencies at build-time. Such results are particularly substantial for deployment and scalability goals in microservices or IoT devices that contain very little memory.

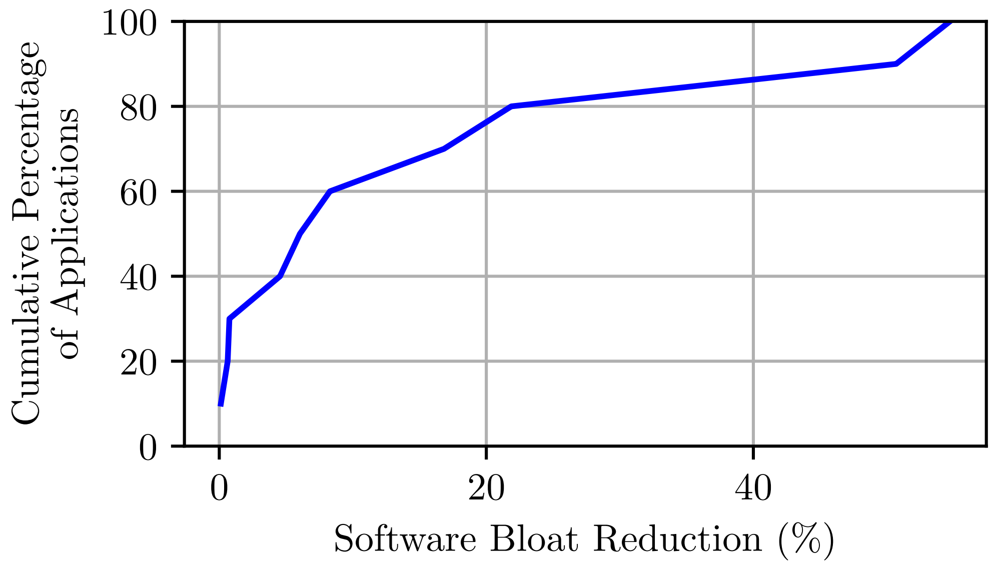


Figure 3.8: Cumulative distribution function (CDF) of software bloat reduction at build-time

3.5.6 RQ6: Performance

As described in Section 3.4, DARCY builds on three tools, Classycle[106], Soot [200], and ANTLR [10]. As a result, to assess DARCY's performance we answer RQ6 in terms of these underlying tools' as well as DARCY's execution time.

Table 3.7: Results for execution time

Component	Avg. Execution Time (ms)
Class Dependency Analyzer (Classycle)	7428
Java Reflection Analysis	328
ServiceLoader Usage Analysis	315
Java Inconsistency Analysis	250
Repair	453
Total	8774

Table 3.7 describes the average execution times for DARCY. Results for Classycle are shown separately from results for other components since Classycle dominates the execution time. On average, DARCY takes under 9 seconds for any system to execute at build-time, which is highly time-efficient for both detection and repair.

3.6 Threats to Validity

In terms of accuracy, the main threat to internal validity is the risk of false positives or negatives of the static analysis tools used in the implementation. False positives or negatives in the results of the static analysis tools may cause DARCY to miss some inconsistencies in the detection phase or report false inconsistencies, which may lead to compilation errors or harming functionality of the application after the repair phase. Since DARCY takes Classycle's results as an input for the Java inconsistency analysis, it inherits all of Classycle's limitations. The accuracy of detecting the inconsistent dependencies is affected by the accuracy of the static analysis tool we use. However, Classycle has been used and in development for over years and leveraged by other state-of-the-art tools for software architecture and antipattern analysis [184, 220, 89, 147, 114, 150, 88]. A similar threat to internal validity exists for our use of Soot; however, Soot is a widely used[130, 144] and actively maintained framework [19] for static analysis of Java programs. we further manually determine whether every identified inconsistency is correct to ensure that any unforeseen issues with underlying static analyses do not compromise DARCY's accuracy.

One of the main threats to external validity is the selection and number of Java applications in the evaluation dataset. To mitigate this threat, we selected open source Java 9+ applications from many developers and about a hundred pages of search results on GitHub, one of the largest and most widely used open-source repositories online. Another threat to external validity is whether the types of inconsistencies we identify comprehensively cover those that may exist. To alleviate this threat, we considered the architectural inconsistencies based on all types of module directives defined in Java 9+.

Another threat to external validity is that DARCY may identify some architectural inconsistencies which was intentionally defined by developers, e.g., in case of Java libraries a developer may export some packages that might not be used internally by the library itself. In these situations, DARCY cannot be aware of the developers or architects' intention and they may disagree with DARCY's repairs. To mitigate this threat, we keep the developers in the loop and let them override DARCY's decision before the repair phase, as explained in Section 3.4.

DARCY's evaluation on only one programming language, i.e., Java, is another threat to external validity. This threat is alleviated by the fact that Java is one of the most widely used languages in the world [22, 21]. Furthermore, the general idea behind DARCY can be extended to any other languages with modular programming constructs that utilize provides and requires interfaces advocated by software architecture-based development and design [202, 162, 146].

3.7 Related Work

The most closely related literature in this area bridges the gap between the prescriptive and descriptive architecture. There are a variety of different types of strategies to address this issue: focusing only on the descriptive architecture by reverse engineering it; obtaining the descriptive architecture and the prescriptive architecture, followed by checking their conformance; ensuring that early in the software lifecycle that the descriptive and prescriptive architectures conform by providing architectural constructs in code; and approaches that ensure conformance of the descriptive and prescriptive architecture from the beginning and into maintenance.

Many approaches address the architecture-implementation mapping issue by ignoring the prescriptive architecture and simply trying to obtain the most accurate descriptive architectures possible [105, 140, 150, 133, 117, 114, 165, 183, 90, 175]. A large number of these approaches rely on software clustering to determine components from implementations [187, 160, 105, 140, 71]. More recently, Hammad et al. [126] have developed a tool-assisted approach that relies on component recovery techniques and converts object-oriented Java apps, i.e., pre-Java-9, to component-based Java 9+ apps that properly use the JPMS constructs with the least privileged architecture specification.

A series of approaches detect inconsistencies between architecture and implementation by reverse engineering the descriptive architecture from the code and comparing it with the prescriptive architecture[168, 161, 201, 169, 197, 55, 219, 180, 100, 182, 142, 86, 141, 124]. Murphy et al. introduced the software reflexion method which helps an engineer compare prescriptive and descriptive architectures in a manual manner [168]. A number of these approaches extend the reflexion method with automated architecture recovery techniques [142, 86, 141]. Moreover, Buckley et al. [77] proposed JITTAC, a tool that uses a real-time Reflexion Modeling approach to provide developers with Just-In-Time architectural consequences of their implementation actions and promote consistency between the prescriptive and descriptive architecture.

Other approaches provide implementation-level constructs that represent architectural elements (e.g., customizable programming-language classes representing components) that help ensure architectural conformance from a forward-engineering perspective [57, 79, 121, 154, 59, 176, 186, 198]. Many of these approaches support various notions of software architectural connectors or interfaces, rather than just components.

Certain approaches achieve architecture-implementation mapping from both a forward-engineering (e.g., code generation) and reverse-engineering perspective, i.e., round-trip engineering [227, 226, 188]. 1.x-way mapping [227] allows manual changes to be initiated in the architecture and a separated portion of the code, with architecture-prescribed code updated solely through code generation. 1.x-line mapping [226] extends 1.x-way mapping to product-line development. Song et al. [188] introduce a runtime approach for architecture-implementation mapping from a roundtrip-engineering perspective.

Another series of approaches focus on detecting and fixing different types of dependency conflicts in software systems, e.g., conflicts in referencing third-party libraries [206, 208, 145, 207, 205, 204, 148]. Wang et al. focus on the semantic conflicts caused by the dependency conflicts and proposed an automated testing technique that attempts to identify inconsistent behaviors of the APIs in conflicting library versions [208] as well as creating the corresponding stack trace [207]. They also proposed techniques to monitor dependency conflicts for Python library ecosystem [205] and detect dependency conflicts in Golang, as well as suggesting proper fixes [204]. Furthermore, Lambers et al. [145] developed a technique that detects all dependency conflicts on multiple granularity levels based on graph transformation. Additionally, Li et al. [148] proposed an automated technique to repair dependency issues where changes in software systems violate package dependency constraints.

Despite other techniques, DARCY (described in chapter 3.4) is the first approach that supports architectural-implementation conformance checking in a mainstream programming language using architectural constructs built directly into the programming language by its creators. Furthermore, our approach includes repairing non-conforming architectures rather than just determining inconsistencies. In addition, DARCY is the only approach for architecture-implementation mapping that focuses on software bloat and attack-surface reduction.

The only existing framework similar to JPMS is OSGI [3]. The major differences between OSGI and JPMS are as follows. OSGI was not able to modularize the JDK, preventing the construction of customized runtime images with a minimized JDK, which JPMS enables. Additionally, OSGI cannot handle reflective access to modules' internal packages. Similar dependency-analysis facilities for OSGI are limited to removing unused dependencies of type import, which represents the require dependency, and cannot cover the other 7 types of inconsistencies in JPMS applications previously introduced in section 3.3. Therefore, there is no similar facility for OSGI that repairs all types of inconsistent dependencies as DARCY does.

3.8 Discussion

This chapter formally defines 8 types of architectural inconsistencies in Java 9+ applications and introduces DARCY, an approach for automatic detection and repair of these types of inconsistencies. DARCY leverages custom static analysis, state-of-the art static analysis tools, and a custom parser generator in its implementation to effectively detect and robustly repair architectural inconsistencies.

The results of our evaluation indicates a pervasive existence of architectural inconsistencies among open source Java 9+ applications. According to our experiment, DARCY's automatic repair results in a significant reduction of the attack surface, enhancement of encapsulation, and reduction of memory usage for Java 9+ applications. In the future, we aim to expand DARCY to other programming languages and improve it to (1) provide architectural visualization and (2) be used as a plug-in for Java Integrated Development Environments (IDE) which helps developers avoid architectural inconsistencies when developing Java applications.

Chapter 4

Dynamic Delivery Testing in Android

Android is a highly fragmented platform with a diverse set of devices and users. To support the deployment of apps in such a heterogeneous setting, Android has introduced *dynamic delivery*—a new model of software deployment in which optional, device- or user-specific functionalities of an app, called *Dynamic Feature Modules (DFMs)*, can be installed, as needed, after the app's initial installation. This model of app deployment, however, has exacerbated the challenges of properly testing Android apps. In this paper, we first describe the results of an extensive study in which we formalized a defect model representing the various conditions under which DFM installations may fail. We then present DELTADROID—a tool aimed at assisting the developers with validating dynamic delivery behavior in their apps by augmenting their existing test suite. Our experimental evaluation using real-world apps corroborates DELTADROID's ability to detect many crashes and unexpected behaviors that the existing automated testing tools cannot replicate.

4.1 Introduction

The Android framework runs on diverse types of devices that vary both in terms of hardware properties, e.g., CPU architectures or hardware sensors, and software configuration, e.g., Android version or default language. Due to this variability, developers need to customize features or offer optional features for specific devices. More importantly, to ensure an app runs properly on different devices, developers need to include various device-specific resources and features in their apps. This strategy forces users to download larger apps with unnecessary code and resources specific to other devices. Alternatively, developers can maintain and publish multiple device-specific APKs for a single app, requiring them to spend additional effort to build, sign, upload, and manage several APKs.

To address these issues, Android has recently introduced a new publishing format, called *Android App Bundle* [29]. One of the main goals of app bundles is to enable *dynamic delivery* [46], i.e., enhance modularization so that developers can customize their apps based on user requirements and deliver optional features on a user's demand. To that end, each app bundle consists of several cohesive modules that together implement an app's full functionality. Specifically, the *base module* of the app bundle implements the core functionality of the app, *configuration modules* include device-specific resources, and *Dynamic Feature Modules* (DFMs) implement optional features. When users install an app for the first time, they install only the base module and a configuration module specific to their devices. DFMs can be downloaded later based on the user's demand[1]. To install a DFM, the app's base module should send an *installation request* to the Android app store operator through its runtime interface, i.e., the app store API. Once the app store operator accepts the request, the app downloads the requested DFM and installs it on the user's device. The lifecycle of an installation request is not always straightforward. In fact, such requests may *fail* due to several reasons and result in critical failures, e.g., crashes [47]. Android documentation has

[1]Developers define how and when users can download different dynamic features on devices running Android 5.0 (API level 21) or higher.

provided instances of best practices regarding dynamic delivery [49], suggesting that developers monitor the state of an installation request throughout its lifecycle and transparently communicate to the user if a failure happens. More specifically, if a DFM installation fails, Android best practice suggests that developers notify the user about the root cause of the failure and adjust the app's behavior.

As a result, to ensure an app's proper behavior in dynamic delivery, developers require tests that (1) reach the installation request in the code and (2) induce the contexts in which the installation request fails. The challenge here lies in the fact that such failures happen under peculiar conditions, e.g., network disconnection, missing the proper permission, or no support for the app store API on the requesting device. Since these conditions cannot be effectively produced through GUI actions, most existing Android testing techniques (e.g., [39, 158, 123]) that target GUI testing are not suitable for testing an app's dynamic delivery. While Android documentation provides a list of conditions that may cause an installation request to fail [46], the description of those conditions is both vague and incomplete, making developers struggle with dynamic delivery testing and understanding the conditions under which dynamic delivery fails [36, 27, 37, 34, 26, 35, 28, 23, 12, 13].

To overcome these challenges, we propose DELTADROID, an approach for **D**ynamic D**EL**ivery **T**esting of **A**n**Droid**, i.e., a test-suite augmentation approach that leverages static and dynamic analysis techniques to (1) identify tests that reach DFMs' installation requests, and (2) modify the identified tests with a combination of GUI and system actions to generate new test cases that induce different conditions, which may cause a DFM installation request to fail. To identify proper actions, it relies on a novel defect model that formally describes the contexts in which failures occur. Such tests will help developers with the steps that lead to DFM installation failure, and they only need to specify application-specific assertions.

Given the fact that Dynamic Delivery has been widely used in recent Android apps [30] (over 40% of all app releases on Google Play [45]), validating the correctness of Dynamic Delivery is crucial. Our evaluation shows that while state-of-the-art [123] and state-of-the-practice [39]

Android testing techniques, which mostly focus on GUI testing, are potentially able to generate tests that can initiate the installation of DFMs, they are not capable of creating the proper conditions to make an installation request fail. More specifically, we detected 44 critical dynamic delivery failures, i.e., crashes, across 25 subject apps. After reporting these failures to developers, 18 of them were verified to date.

This chapter makes the following contributions:

- Construction of a defect model representing Android dynamic delivery failure through the formalization of failure conditions.
- A hybrid static and dynamic analysis technique for augmenting existing test suites with tests that can validate Android dynamic delivery. An implementation of the proposed technique is publicly available [51].
- An extensive empirical evaluation on real-world Android apps demonstrating DELTADROID's effectiveness in test augmentation.

The remainder of this chapter is organized as follows. Section 4.2 illustrates a motivating example. Section 4.3 introduces the formal specifications of the defect model. Section 4.4 describes the details of DELTADROID and its implementation. Section 4.5 presents the experimental evaluation of the research. The chapter concludes with a discussion of the related research and avenues of future work.

4.2 Illustrative Example

As an illustrative example, we use the Android app K9Mail [31]. We describe the functionality of K9Mail, a defect in the app associated with dynamic delivery, and how a test can reveal the defect.

App: K9Mail is an email client app whose main functionalities are viewing and sending emails, managing multiple email accounts, searching, etc. In addition to these core functionalities, K9Mail offers an optional feature for encrypting emails to provide an additional level of security. Since most users do not use this feature, it is encapsulated into a DFM which users can install on-demand.

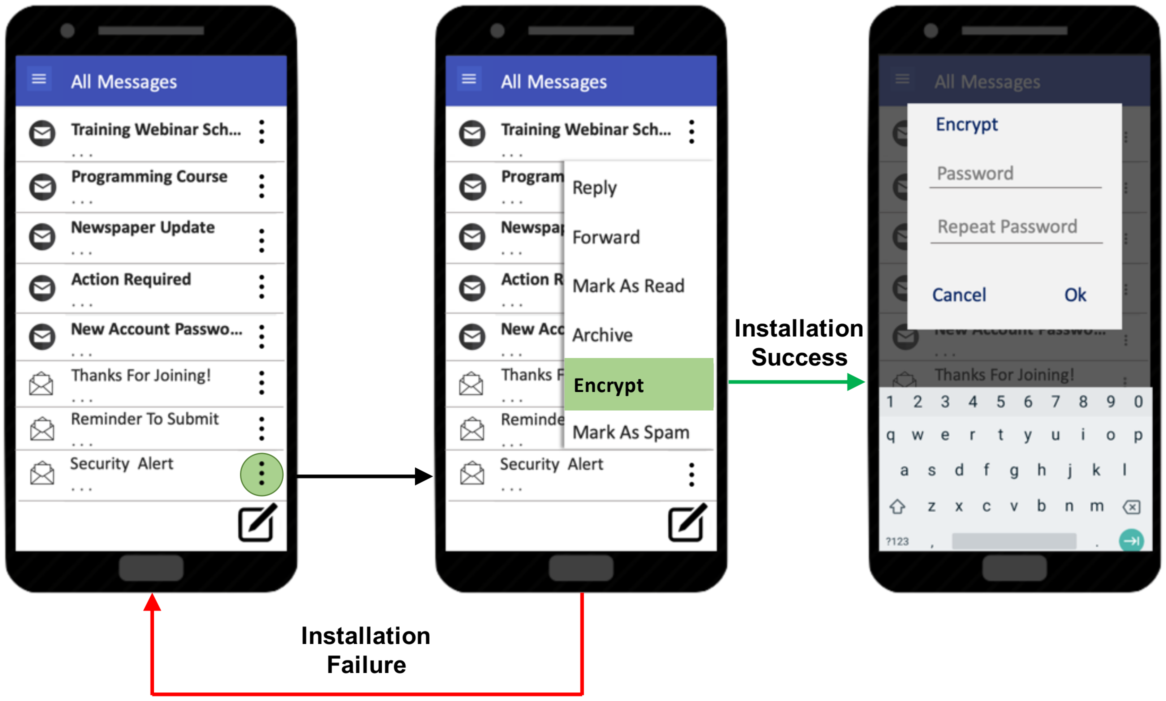


Figure 4.1: Installation of a new DFM on K9Mail

Defect Scenarios: Figure 4.1 shows three steps that lead to installation of the *Encryption* DFM. First, the user clicks on an email's menu—denoted by three dots—to access more options. Clicking on the "Encrypt" option initiates the installation of the *Encryption* DFM. If the module installation is successful, the user will see a dialog to enter a password and confirms the encryption. Otherwise, if the installation request fails due to a contextual reason (Section 4.3), K9Mail goes to the initial screen without notifying the users about the failure or instructing them to take proper actions—potentially giving users the false impression that the email is encrypted. Without a proper test to reveal the failure scenario, a developer may not detect this defect. We describe two possible dynamic

```
#click on the "Security Alert" email's menu icon
driver.find_element_by_xpath(path_to_icon).click()
#click on the "Encrypt" option
driver.find_element_by_xpath(path_to_encrypt).click()
#enter encryption password in the password field
driver.find_element_by_id(password_field_ID).
     send_keys(password)
#repeat password
driver.find_element_by_id(password_confirm_field_ID).send_keys(password)
#click on "OK" button
driver.find_element_by_id(OK_button_ID).click()
```

Figure 4.2: An Appium test, written in Python, to test the installation success scenario of Figure 4.1

```
#click on the "Security Alert" email's menu icon
driver.find_element_by_xpath(path_to_icon).click()
#disconnect network connection
driver.set_network_connection(0)
#click on the "Encrypt" option
driver.find_element_by_xpath(path_to_encrypt).click()
#restore network connection
driver.set_network_connection(1)
# check if network connection alert dialog is displayed
Assert.assertFalse(driver.find_element_by_id(Network_Alert_Dialog_ID).isEmpty())
```

Figure 4.3: An Appium test, written in Python, to test the installation failure scenario of Figure 4.1 due to no network connection

```
#click on the "Security Alert" email's menu icon
driver.find_element_by_xpath(path_to_icon).click()
#Filling the storage of the device with a huge file
args = {"command": "df","args": ["/sdcard", "|", "grep", "\'data\'", "|", "awk", "-F", "\' \'
     ", "\'{print $4}\'"]}
bytesAvailable = driver.execute_script("mobile: shell", args)
args = {"command": "dd","args": ["if=/dev/zero", "of=/sdcard/deleteme", "bs=1B", "count=" +
     str(int(bytesAvailable))]}
result = driver.execute_script("mobile: shell", args)
#click on the "Encrypt" option
driver.find_element_by_xpath(path_to_encrypt).click()
#Deleting the huge file to free up the storage
args = {"command": "rm","args": ["/sdcard/deleteme"]}
driver.execute_script("mobile: shell", args)
# check if insufficient storage alert dialog is displayed
Assert.assertFalse(driver.find_element_by_id(Insufficient_Storage_Alert_Dialog_ID).isEmpty())
```

Figure 4.4: An Appium test, written in Python, to test the installation failure scenario of Figure 4.1 due to insufficient storage

delivery failures for this scenario, i.e., network disconnection [25] and insufficient storage [38]. We further discuss properties of the required tests to identify such failures.

Tests: Figure 4.2 shows an Appium [40] system test to validate the behavior of K9Mail during installation of the *Encryption* module. While this test can validate proper implementation of K9Mail when the installation request is successful, it is unable to capture dynamic delivery failures occurring due to network disconnection or insufficient storage. Consequently, we need two additional tests to validate proper handling of unsuccessful installation of the *Encryption* module. These new tests, shown in Figure 4.3 and Figure 4.4, include adding system events to the test in Figure 4.2. The test in Figure 4.3 disconnects the network connection before clicking on the “Encrypt” option (line 4), making the installation request fail, and then re-connects the network connection (line 8) after clicking on the “Encrypt” option. The test then checks whether the desired alert dialog, notifying the user about the network connection failure, is displayed. On the other hand, the test in Figure 4.4 fills the storage of the device before clicking on the “Encrypt” option, making the installation request fail due to insufficient storage (lines 4–7), and then revert it after clicking on the “Encrypt” option (lines 11–12). The test then checks if the desired dialog, asking the user to remove unnecessary files, is displayed. There are two main challenges to design such tests:

(1) Installation request failures depend on the contextual settings in which a test is executed and cannot be achieved simply through GUI actions. To that end, developers need to inject additional system or GUI events into the test event sequences to induce a failure. In the test case of Figure 4.3, line 4 contains the system event that disables the network, making the installation request for the *Encryption* DFM fail due to a network error (Section 4.3, Equation 4.1). To ensure that the change in the state of the network only impacts the DFM installation and not the rest of the test, the network connection is restored afterward.

(2) The position of the injected events in the test event sequence is critical. For example, if the developer disables the network connection at the very beginning (line 2 in Figure 4.2), the test cannot load the list of emails, cannot find the menu icon for the “Security Alert” email, and thus fails without being able to execute the module installation, resulting in an invalid test. Similarly, if the network connection is disabled after initiating the installation request (line 4 in Figure 4.2), the test cannot observe the behavior of K9Mail when the installation request fails; therefore, it would not be a useful test for validating the *Encryption* DFM’s behavior.

4.3 Defect Model

The previous section's defect scenario describes an installation request failure. Developers should be aware of all the different situations causing the installation failure, two of which mentioned in Section 4.2, and take appropriate actions to handle them. To identify the situations leading to an installation failure, we first need to describe the process through which a DFM is installed.

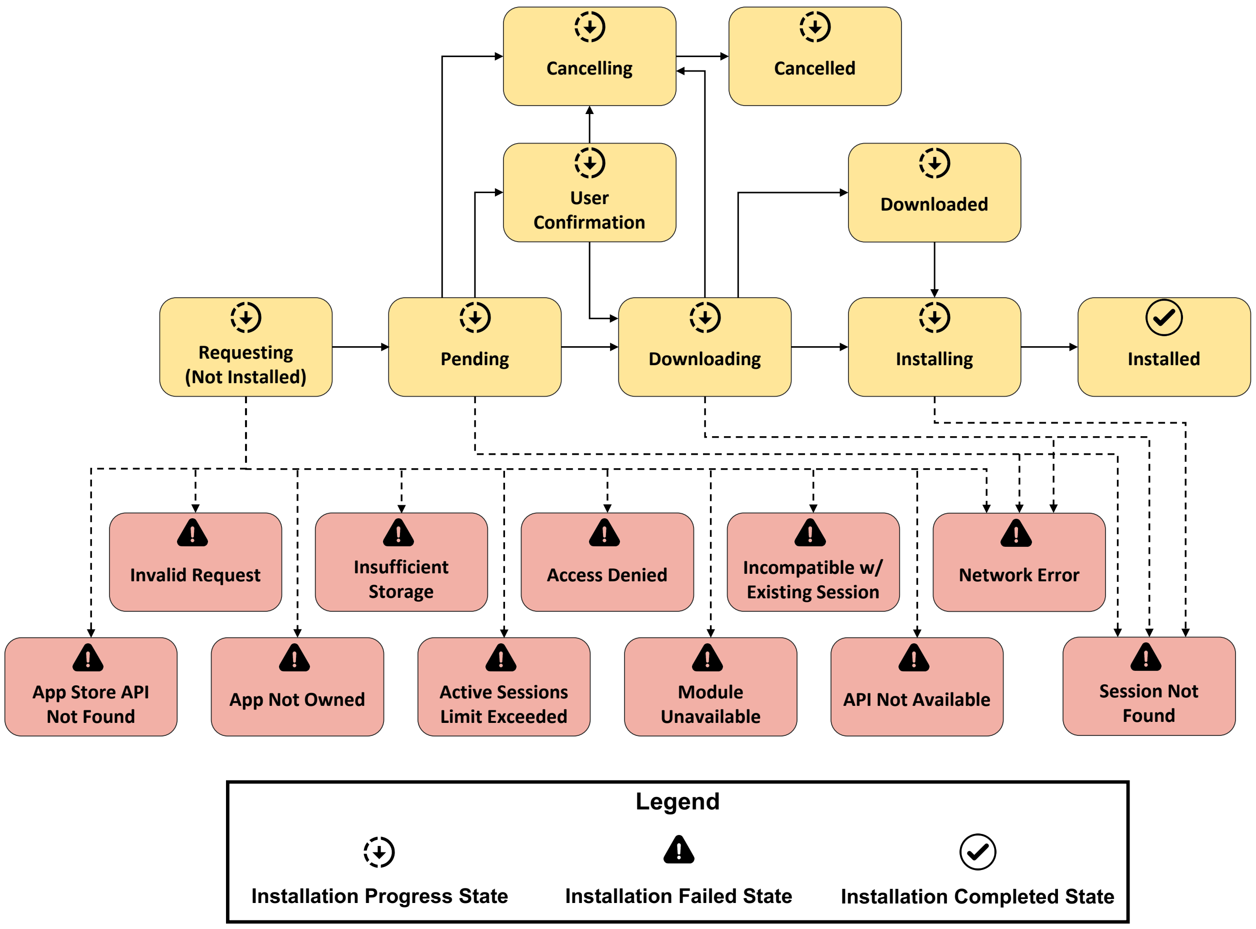


Figure 4.5: The lifecycle of a DFM installation request

Once an Android app attempts to install a DFM, it creates an installation request that can go through multiple states, as demonstrated in Figure 4.5. First, at the *Requesting* state, the app store API—the app's runtime interface with the app store services—sends the installation request to the app store (e.g., Google Play [45]). In the *Pending* state, the request has been accepted by the app store operator, and the download should start momentarily. Through Android dynamic delivery, DFMs are downloaded as separate APK files, called *split APKs*. During download time, the request is in the *Downloading* state. Once the device completes the download, in case the app has SplitCompat[2] installed, it can immediately access the DFM to install it, leading the request to the *Installing* state. Otherwise, the request will go to the *Downloaded* state, where the DFM split APK is downloaded but cannot yet be installed. As soon as SplitCompat is installed in the app, the DFM can be installed in the background by the app store operator. Finally, if the installation finishes successfully, the request is in the *Installed* state. During the *Pending* or *Downloading* states, the installation request can be cancelled, leading to the *Cancelling* and *Canceled* states. After download, the Android device automatically installs the DFM, which can be uninstalled by the user later. For modules that are larger than 10MB, the installation requires user confirmation (the *User Confirmation* state). If the user confirms the installation, the request follows the normal lifecycle. Otherwise, it goes to the *Cancelling* and *Canceled* states [46].

An installation request may also terminate the normal lifecycle and go to any of the *Failed* states, which is denoted by dashed lines in Figure 4.5. Based on the root cause of the failure, developers need to handle it accordingly. For instance, if the request fails due to a *network error*, developers should check the network connection and only resend the request if the network is stable. As another example, if the request fails due to *insufficient storage*, which prevents the module from being installed on the device, the developer should ask the user to free up some space, install the module, and monitor for different root causes of the module installation failure.

While Android documentation briefly explains the *Failed* states and provides recommendations for proper actions to handle failure of an installation request [46], it does not sufficiently describe the context required to reach the *Failed* states for the purpose of testing. In fact, developers might have a hard time understanding the root cause of each *Failed* state [36, 27, 37, 34, 26, 35, 28, 23, 12, 13, 24]. For instance, [24] states, "My application uses app bundle, but dynamic feature performs a very poor rate of success when downloading, with a lot of error codes, and I cannot do anything for

[2]SplitCompat is an Android library class that enables immediate access to code and resources of the DFM split APKs.

it. I search the docs but no help!". To overcome this challenge, we constructed a defect model for Android dynamic delivery failures. This defect model formally defines a *Failed State Context* as the context in which its associated *Failed* state will occur. The formulation will be used by DELTADROID to augment existing tests to induce *Failed* states (Section 4.4).

To construct a comprehensive defect model, we explored multiple resources to understand the root causes of dynamic delivery failures and to identify contextual factors that trigger them. Specifically, we started from Android documentation and classified failures into 11 classes, based on the different error messages Android framework generates upon the occurrence of each failure [47]. We then explored issue repositories on GitHub [11] and discussion forums (e.g., StackOverFlow [48]) for keywords related to installation errors and DFM install failures, such as *dynamic feature module installation failure*, *SplitInstallErrorCode*, names of each specific error code and their messages included in the Android documentation [47]. Additionally, we investigated many open-source Android apps' implementations for instances of `SplitInstallErrorCode` and the way developers handle them using `onFailureListeners` upon the occurrence of different failures. Investigating these resources, we identified the contextual factors contributing to the manifestation of dynamic delivery. For a few cases where the mentioned resources were inconclusive, we exhaustively examined all the relevant contextual factors that could produce the desired *Failed* state.

4.3.1 Failed State Context Definition

The manifestation of a *Failed* state depends on the properties of an app bundle, DFM, installation request, and device configurations.

DEFINITION 4.1. *The configuration of a device in which a DFM will be installed can be formally specified as a tuple*
$Dev \equiv \langle Android\_Version, App\_Store, Account, Network, Storage \rangle$, *where*

- *Android_Version is the device's Android version,*

- *App_Store is the device's app store app (e.g., Play Store app),*

- *Account is the user's app store account registered on the device,*

- *$Network \in \{true, false\}$ is the network connection status of the device (true if connected and false otherwise), and*

- *Storage is the amount of storage available on the device.*

DEFINITION 4.2. *An app bundle can be represented as a tuple $AB \equiv \langle DFMs,\ Permissions,\ DownloadSource\rangle$ where*

- *$DFMs \equiv \{DFM_i \mid i \in \{1,...,n\}\}$ is a set of all DFMs in the app bundle AB,*

- *Permissions is the set of permissions required to install DFMs, and*

- *$DownloadSource$ is the source from where the app bundle has been downloaded, e.g., Google Play Store.*

DEFINITION 4.3. *A DFM can be formally specified as a tuple $DFM \equiv \langle AB,\ Min\_SDK\_Version,\ Authorized\_Users,\ Size\rangle$ where*

- *AB is the app bundle of DFM,*

- *$Min\_SDK\_Version$ indicates the minimum API version of the Android device that the DFM is compatible with, i.e., minSdkVersion in DFM build file,*

- *$Authorized\_Users$ are a set of users' app store accounts that have access to the DFM and can install it, and*

- *Size is the necessary storage for installation of DFM.*

DEFINITION 4.4. *An installation request, i.e., request from an app to install a DFM is defined as a tuple $Req \equiv \langle SessionID,\ DFM,\ State\rangle$ where*

- *SessionID is a unique identifier for a request assigned by the app store API to track the status of the request,*
- *DFM is the dynamic feature module that will be installed through this request, and*
- *$State \in \{R, P, UC, D_{ing}, D_{ed}, I_{ing}, I_{ed}, C_{ing}, C_{ed}, F\}$ is the current state of the installation request. As shown in Figure 4.5, the possible values are R (Requesting), P (Pending), UC (User Confirmation), D_{ing} (Downloading), D_{ed} (Downloaded), I_{ing} (Installing), I_{ed} (Installed), C_{ing} (Cancelling), C_{ed} (Canceled), and F (Failed).*

With these definitions, we can represent a Failed State Context as a tuple $FSC = \langle Dev, AB, Req \rangle$ and identify the contextual settings in which a distinct *Failed* state happens.

4.3.2 Failed States

As shown in Figure 4.5, an installation request may terminate in ten different *Failed* states: *Network Error, Insufficient Storage, Access Denied, Incompatible With Existing Session, Active Sessions Limit Exceeded, Module Unavailable, API Not Available, Session Not Found, App Not Owned*, and *Invalid Request*.

1. Network Error. An installation request goes to Network Error state if a network exception [25] occurs when the request is in any of the *Requesting*, *Pending*, or *Downloading* states. For a $dev : Dev$ and $req : Req$, the contextual parameters that induce such failure are as follows:

$$FSC_{Network\_Error} \equiv req.State \in \{R, P, D_{ing}\} \quad \wedge \quad dev.Network\_Connection = false \tag{4.1}$$

To ensure developers have considered a proper action when such failure happens, there should be a test to disconnect network connection during the lifecycle of the request.

2. Insufficient Storage. The Insufficient Storage failure happens when an app requests to install a DFM, whose size is larger than available storage on the phone. More specifically, once an installation request is initiated, the Android device checks its remaining storage before starting the installation request, and in case of insufficient available storage it fails at the *Requesting* state. It will only start downloading and installing in case the device has enough storage. For a $dev : Dev$ and $req : Req$, the contextual parameters that induce such failure are as follows:

$$FSC_{Insufficient\_Storage} \equiv req.State \in \{R\} \ \wedge \ (req.DFM.Size > dev.Storage) \tag{4.2}$$

Developers should handle this situation, e.g., show a proper notification informing the the user about the space availability issue on the phone. To validate this behavior, a test needs to mock the storage of the device as being full.

3. Access Denied. Without proper permissions to download a DFM, i.e., `REQUEST_INSTALL_PACKAGES` permission, an installation request from the app store operator is denied in its inception, i.e., when in the *Requesting* state. For an $ab : AB$ and $req : Req$, the contextual parameters that induce such failure are as follows:

$$FSC_{Access\_Denied} \equiv req.State \in \{R\} \ \wedge \ Request\_Install\_Packages \notin ab.Permissions \tag{4.3}$$

To ensure developers implement proper actions in response to this failure, test suites should have either a test that permanently revokes the app's `REQUEST_INSTALL_PACKAGES` permission or temporarily revokes it by taking the app to the background.[3]

4. Incompatible With Existing Session. Installation of a DFM might be accessible from different paths in the program. Thus, more than one installation request can be sent to the app store operator for the same DFM. Specifically, consider the case where an installation request of a DFM is in progress and the DFM is not installed yet, and meanwhile, another installation request attempts to install the same DFM. In this

[3]From Android 11, an app's permission can be temporarily revoked while it is not in the foreground.

case, the second request will terminate and reach the *Incompatible With Existing Session* state. This failed state happens while the first request is in any of the *Requesting*, *Pending*, *Downloading*, or *Installing* states, and the second request is in the *Requesting* state. For a $dev : Dev$, $req1 : Req$, and $req2 : Req$, the contextual parameters that induce such failure are as follows:

$$\begin{aligned}
&FSC_{Incompatible\_With\_Existing\_Session} \equiv \\
&req_1.DFM = req_2.DFM \\
&\wedge\ req_1.SessionID\ \neq\ req_2.SessionID \\
&\wedge\ req_1.State \in \{R, P, D_{ing}, I_{ing}\}\ \wedge\ req_2.State \in \{R\}
\end{aligned} \tag{4.4}$$

To ensure the app implements proper means of accounting for this failure, a test should simultaneously initialize at least two installation requests for the same DFM.

5. Active Sessions Limit Exceeded. While the previous situation includes multiple requests for the same DFM, an installation request for a DFM is also rejected if there is already an active installation request for another DFM. This failed state occurs while the first request is in any of the *Pending*, *Downloading*, or *Installing* states, and the second request is in the *Requesting* state. For a $dev : Dev$, $req1 : Req$, and $req2 : Req$ the contextual parameters that induce such failure are as follows:

$$\begin{aligned}
&FSC_{Active\_Sessions\_Limit\_Exceeded} \equiv \\
&req_1.DFM \neq req_2.DFM\ \wedge \\
&req_1.State \in \{P, D_{ing}, I_{ing}\}\ \wedge\ req_2.State \in \{R\}
\end{aligned} \tag{4.5}$$

If the app contains multiple DFMs in the bundle, installing any of them may result in *Active Session Limit Exceeded* failed state. Since the implementation of these DFMs might be distributed in the code, different test cases executing different parts of the code are required to induce this failed state.

It is worth to note that *Active Session Limit Exceeded* state does not subsume *Incompatible with Existing Session*, as the *SessionID* of req_1 and req_2 might be the same in the former. To ensure developers implement proper actions to account for this failure, a test should simultaneously initialize installation requests for different DFMs.

6. Module Unavailable. This failed state can occur due to two possible causes: (1) For each DFM installation request, the app store API checks whether the `minSdkVersion` of the DFM matches that of the Android system running on the device. *Module Unavailable* failed state occurs if the Android version running on the device is less than the `minSdkVersion` in the DFM's build file, i.e., mismatches the required API Version of the DFM. (2) In some cases, the developer might not make the DFM available to all users. If a DFM is unavailable for the user's app store account on the device, the request reaches the *Module Unavailable* state. This failed state happens while the request is in the *Requesting* state. For a *dev* : *Dev* and *req* : *Req*, the contextual parameters that induce such a failure are as follows:

$$\begin{aligned} &FSC_{Module\_Unavailable} \equiv req.State \in \{R\} \wedge \\ &(\, dev.Android\_Version < req.DFM.Min\_SDK\_Version \vee \\ &dev.Account \notin req.DFM.Authorized\_Users\,) \end{aligned} \tag{4.6}$$

To validate the app implements proper means of handling this failure, a test suites should include a test that either changes the Android version of the device or the app store account on the device.

7. API Not Available. App bundles and thus on-demand DFMs are available on Android versions 5 (API level 21) or higher. For older versions, developers should change the configuration of DFMs, such that they can be downloaded with the base APK. This allows the app to support downloading all modules of an APK, similar to a monolithic APK from Android versions that do not support DFMs. Otherwise, users of devices with older versions can only download the base APK. Attempts of an app to violate this by trying to download a DFM results in the installation request terminating with the *API Not Available* error. This failed state occurs while the request is in the *Requesting* state. For a *dev* : *Dev* and *req* : *Req*, the contextual parameters that induce such failure are as follows:

$$FSC_{API\_Not\_Available} \equiv \quad req.State \in \{R\} \wedge dev.Android\_Version < Android\_5 \tag{4.7}$$

A test to validate the proper action to account for this failure should mock the Android version running on the device.

8. App Store API Not Found. This *Failed* state happens when the app store (e.g., Play Store app) is either not installed or not the official version. This failed state occurs while the request is in the *Requesting* state. For a $dev : Dev$ and $req : Req$, the contextual parameters that induce such failure are as follows:

$$FSC_{App\_Store\_API\_Not\_Found} \equiv req.State \in \{R\} \wedge dev.App\_Store \neq Official\_App\_Store\_app \tag{4.8}$$

A test to validate an app's behavior in response to this failure should mock a device that does not have the official app store installed.

9. Session Not Found. This *Failed* state happens when session ID generated by the app store operator for an installation request is not valid, i.e., does not correspond to an active session in the app store services. This failed state occurs while the request is in any of the *Pending*, *Downloading*, or *Installing* states. For a $req : Req$, the contextual parameters that induce such failure are as follows:

$$FSC_{Session\_Not\_Found} \equiv req.State \in \{P, D_{ing}, I_{ing}\} \wedge req.SessionID = invalid \tag{4.9}$$

10. App Not Owned. This *Failed* state happens when the app has not been installed by the designated app store and the feature cannot be downloaded. This failed state occurs while the request is in the *Requesting* state. For a $req : Req$, the contextual parameters that induce such failure are as follows:

$$FSC_{App\_Not\_Owned} \equiv req.State \in \{R\} \wedge ab.DownloadSource \neq Official\_App\_Store\_app \tag{4.10}$$

11. Invalid Request. This *Failed* state happens when the app store received the installation request, but the request is not valid., i.e., the information included in the request is not complete or accurate. This failed state occurs while the request is in the *Requesting* state. For an $ab : AB$ and a $req : Req$, the contextual parameters that induce such a failure are as follows:

$$FSC_{Invalid\_Request} \equiv req.State \in \{R\} \wedge \ (req.SessionID = invalid \ \vee req.DFM = invalid) \tag{4.11}$$

A test to validate an app's behavior in response to the last three failures should mock the app store operator API to generate an invalid session ID for an installation request, e.g., $SessionID = null$, or an invalid source of download for an app bundle.

4.4 DELTADROID: Framework Overview

Figure 4.6 depicts a high-level overview of DELTADROID, consisting of two major components: *Test-Suite Analysis* and *Test-Suite Augmentation*.

Test-Suite Analysis takes an Android app bundle and the corresponding initial test suite as inputs and produces the *Baseline Test Cases*—representing those tests that initiate a DFM installation—and the associated *Metadata*—data about the specific test step in which a DFM is installed. The *Baseline Test Cases* are subsequently reused in the creation of new tests.

Test-Suite Augmentation takes the *Baseline Test Cases* and associated Metadata together with the *Defect Model*, which formally defines the contexts in which a DFM installation request can fail, as inputs and generates additional tests that are effective for dynamic delivery validation. Specifically, this component creates new tests by first replicating a baseline test and then modifying it with a combination of system and GUI events to induce the context that may manifest installation request failures. The newly generated tests create all the failure scenarios that an app should handle. In the remainder of this section, we provide a more detailed explanation of the components comprising DELTADROID.

4.4.1 Test-Suite Analysis

The *Test-Suite Analysis* component identifies tests that are suitable for validating the behavior of dynamic delivery in the app. To that end, *App Bundle Analyzer* pinpoints the code responsible for installing DFMs in the app and instruments it. Subsequently, *DFM Install Detector* executes the initial test suite on the instrumented app to determine the *Baseline Test Cases*, i.e., the tests in the initial test suite that execute the installation request. Finally, *Test Analyzer* identifies the locations in baseline tests that additional GUI or

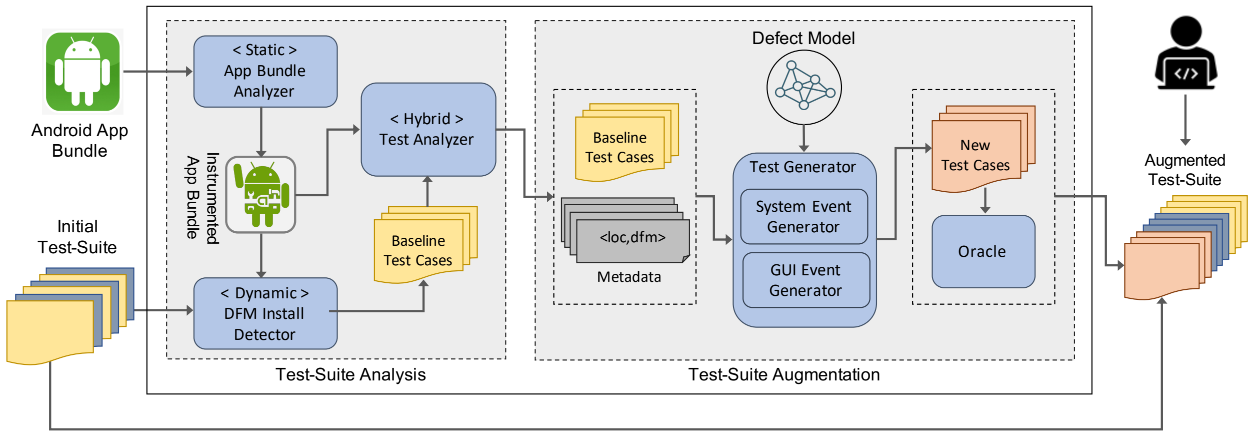


Figure 4.6: A high-level overview of DELTADROID

system events should be injected to create effective dynamic delivery tests.

App Bundle Analyzer.

To aid in understanding how *App Bundle Analyzer* works, Figure 4.7 shows the code snippet corresponding to the installation request of the `Encryption` DFM in k9Mail. The code initiates the request by creating instances of `SplitInstallManager` (line 2) and `SplitInstallRequest` (line 4). It then sends the installation request of the `Encryption` DFM to the app store API by invoking the `startInstall` method (line 6).

App Bundle Analyzer performs a flow-sensitive analysis to find invocations of `startInstall` method and injects a logging statement immediately before it in the control-flow graph. Additionally, *App Bundle Analyzer* registers a `SplitInstallStateUpdatedListener` to monitor and log the state of the installation request at run-time.

```
1 //create an instance of SplitInstallManager.
2 SplitInstallManager splitInstallManager = SplitInstallManagerFactory.
      create(context);
3 //create a request to install a module.
4 SplitInstallRequest request = SplitInstallRequest.newBuilder().addModule
      ("Encryption").build();
5 //submit the request to install the module
6 splitInstallManager.startInstall(request);
```

Figure 4.7: Code snippet from K9Mail app that requests the installation of *Encryption* DFM

DFM Install Detector.

DFM Install Detector takes the instrumented app bundle as input, runs it against the initial test suite, and identifies the tests that initiate DFM installation requests as *Baseline Test Cases*, i.e., tests that execute the instrumented log statements.

Test Analyzer.

After identifying *Baseline Test Cases*, the next step is to find the proper location of modification, i.e., the index in the test event sequence where new test events should be injected, which we refer to as *target location*. To that end, *Test Analyzer* first instruments the tests to record all the test events, their types, and orders. Events could be either GUI events (e.g., button clicks) or system events (e.g., changes in network connection). It then uses Algorithm 1 to locate the last event in each test after which an installation request is initiated, i.e., the *target location*.

Algorithm 1 shows how *Test Analyzer* identifies target locations. *Test Analyzer* takes the instrumented *app* and a set of baseline tests, T, as input, and provides a set of tuples $\langle$ *location*, *DFM*$\rangle$ as output. The first element in the output tuple, $location$, indicates the index at which a $test \in T$ should be modified, and the second element indicates the DFM that is installed by $test$ at that $location$.

Algorithm 1 starts by initializing the metadata, *MD*, to an empty set. For each instrumented test $test \in T$, Algorithm 1 executes the test on the instrumented app bundle, app, and collects all the entries recorded in the log file during test execution in $loggedEntries$ (line 3). These entries are either test events that are recorded by the instrumented test, $test$, or log messages that show app has initiated an installation request. At the next step, the Algorithm sorts the logged entries based on their corresponding time stamp, i.e., the actual time the event happens, to account for potential race conditions (line 4). Subsequently, Algorithm 1 iterates over all the entries in $loggedEntries$ to identify the target location, i.e., the location of the last event in each baseline test after which an installation request is initiated (lines 5-13). Specifically, it starts from the first entry in $loggedEntries$ and keeps track of the order of test events (lines 6-7). Once Algorithm 1 reaches an entry that is made by app that installs a DFM (line 8), it extracts the DFM identifier from the entry (line 9) along with

```
Algorithm 1: Test Analyzer
    Input: $app$  // Instrumented app bundle
    Input: $T$  // Baseline tests
    Output: MD: $\{md \mid md = \langle location, DFM \rangle\}$  // Baseline tests' metadata
 1  $MD \leftarrow \emptyset$
 2  foreach $test \in T$ do
 3      $loggedEntries \leftarrow$ RunTestCase($test, app$)
 4      sortByTimeStamp($loggedEntries$)
 5      foreach $entry \in loggedEntries$ do
 6          if $entry.type$ is TESTEVENT then
 7              $location \leftarrow$ getEventID($entry$)
 8          else
 9              $DFM \leftarrow$ getDFMIDentifier($entry$)
10              $MD \leftarrow MD \cup \langle location, DFM \rangle$
11              break
12          end
13      end
14  end
```

the location of the last test event, *location*, saves it as a tuple in *MD* (line 10) for the corresponding test.

The algorithm terminates once the target locations corresponding to all tests are identified. As a result, it provides a set of tuples containing the information required by the next component in DELTADROID's workflow, i.e., *Test-Suite Augmentation*.

4.4.2 Test-Suite Augmentation

Test-Suite Augmentation takes our defect model (Section 4.3) as well as the output of *Test-Suite Analyzer* as inputs and generates effective dynamic delivery tests as output, i.e. 11 additional tests for a given baseline test. Each of these new tests is responsible for creating one of the *FSC*s formally defined in Section 4.3 to make the installation requests fail. *FSC*s can be created by injecting additional system and GUI events into the test event sequence.

Algorithm 2 shows how *Test-Suite Augmentation* generates effective dynamic delivery tests. There are three inputs to this algorithm. The first input is $FSCs = \{fsc \mid fsc = \langle Dev, AB, Req \rangle\}$, which is a set of *Failed State*

Contexts we formally defined in Section 4.3.2. The second and third inputs are T and MD, respectively, the output of *Test-Suite Analysis*. T represents a set of candidate tests and MD is the corresponding metadata, i.e., information about the DFM each test installs and the location at which DFM installation occurs within each baseline test.

Algorithm 2 initializes *GenTCs*, a set of newly generated tests, as an empty set (line 1). It then iterates over baseline tests (lines 2-7) and for each DFM installation request by a baseline test, it generates 11 new tests, where each of them is responsible to create the context corresponding to one of the FSCs (lines 3-6). Specifically, Algorithm 2 generates a set of test steps for creating system events (line 4) and GUI events (line 5), and injects them in the baseline *test* to create a new test, $test'$, (line 6). After generation of each new test, Algorithm 2 updates *GenTCs* with the new test (line 7).

Algorithm 2: *Test-Suite Augmentation*

```
   Input: FSCs: ${fsc \mid fsc = \langle Dev, AB, Req \rangle}$  // Failed State Context formula
   Input: T // Instrumented baseline tests
   Input: MD: ${md \mid md = \langle location, DFM \rangle}$ // Metadata corresponding to baseline tests
   Output: GenTCs
1  $GenTCs \leftarrow \emptyset$
2  foreach $test \in T$ do
3      foreach $fsc \in FSCs$ do
4          $events_{sys} \leftarrow$ systemEventGenerator($fsc$)
5          $events_{GUI} \leftarrow$ GUIEventGenerator($fsc$)
6          $test' \leftarrow test \cup$ injectEvents($md, events_{sys}, events_{GUI}$)
7          $GenTCs \leftarrow GenTCs \cup test'$
8      end
9  end
```

The `systemEventGenerator` method is responsible for generation of proper system events. It takes the formulation of a *Failed State Context,fsc*, parses the formulation to obtain contextual properties that contribute to the corresponding *fsc*'s failure, and generates system events accordingly. For example, to generate the example test for k9Mail (Figure 4.3), `systemEventGenerator` parses $FSC_{Network\_Error}$ and identifies the state of the request, *req.State*, and connectivity status of the device, *dev.Network_Connection*, as the related contextual parameters (Equation 4.1) and generates corresponding system events. There are two categories of system events that `systemEventGenerator` injects into the baseline tests. The first category includes system events that directly change a context, e.g., disabling the network connection, slowing down the network speed, or creating a large file to take up device storage. The second category includes system events that indirectly change contexts by simulating the device's running Android version, user's app store account, or the *SessionID* assigned to an installation request.

For a subset of FSCs in which the failure originates in the app store API (i.e., Equations 4.6-4.11), `systemEventGenerator` mocks the app store API to induce a specific behavior. For instance, in *Module Unavailable FSC*, the failed state will be induced if the version of the requested module is not compatible with the app bundle uploaded in the app store. To that end, `systemEventGenerator` replaces `SplitInstallManager` with a customized implementation reflecting the required context.

System events and mocking are not enough for creating a subset of FSCs. For instance, in the case of *Access Denied* (Formula 3), an effective test should revoke the `REQUEST_INSTALL_PERMISSION` of the app. This can be done either by permanently revoking the permission through Android settings or temporarily revoking the app's permission to download a DFM by moving the app to the background right before the DFM installation begins. Each of these decisions requires a series of consecutive GUI actions. For example, one series of GUI actions may involve clicking on the *Home* button to move the app to the background, relaunching the app in the same state by clicking the *App Switch* button, and choosing the app among the recent running apps.

After producing the proper events, Algorithm 2 injects them into a copy of the given baseline test and generates a new test (line 6). To that end, the `injectEvents` method takes $event_{sys}$ and $event_{GUI}$ sets and injects them into $test$ at the index specified by $md.location$. Specifically, `injectEvents` first injects $event_{sys}$ and then $event_{GUI}$. This is mainly because some GUI actions are useless without proper system events before them. For example, in the case of *Active Session Limit Exceeded*, the installation request should be in the *Pending*, *Downloading*, or *Installing* states *before* requesting installation of another DFM to make the failure occur. To ensure this, we need to inject system events that slow down the speed of network connection, making the transition of installation requests between states slower and allowing GUI actions to be effective. In the rare cases where the baseline tests already contain proper events to simulate any or a subset of FSCs, the specific FSC(s) are simply repeated. This repetition does not break the augmented test.

Finally, once a new test is generated, Algorithm 2 adds it to the set of newly generated tests. Algorithm 2 terminates after generating all the tests that make each installation request enter all the 11 *Failed* states described in Section 4.3. That is, the size of *GenTCs* is 11 times the size of baseline tests, i.e., $|GenTCs| = 11 \times |T|$.

Once the new tests are generated, the *Oracle* component automatically determines if developers properly handle the failed states. We follow the specification of *unhandled* behavior from Android documentation [47], in which an app takes no action, visible to the user, to handle the failure. Such behavior in our illustrative example (Section 4.2) could be due to either missing an `OnFailureListener` in the code or any additional UI element on the app screen notifying the user about a failure. To identify *unhandled* behaviors, the oracle performs two actions. First, it investigates the invocation of `OnFailureListener` during the execution. If such a callback is not invoked after an installation failure, a clear indicator is that a DFM failure is not handled in the implementation. However, a developer may implement `OnFailureListener` in the code without actually handling the failure in a meaningful manner. To account for such cases, our proposed oracle investigates changes in the visible UI elements *before* and *after* the DFM installation failure. If there is an addition of UI elements to the screen, e.g., a UI widget is added to show an error message, developers have satisfied the minimum requirement to handle the DFM failure, i.e., informing users about the failure [47]. Otherwise, the oracle can confidently judge that the developer has not properly handled the DFM installation failure.

We do not generate assertions here for two reasons: First, the purpose of DFM testing is to help developers with the steps that lead to dynamic delivery failures and determine whether a particular failed state was handled properly. Assertions for dynamic delivery testing are highly app-specific. For example, depending on the developer's design decision, they can either show a message, which notifies the user about the failure and its reason, or stops the installation progress bar. In the former case, the assertion checks for the visibility of a dialog message, e.g., line 10 in Figure 4.3. In the latter, the assertion could check for a specific amount of progress (e.g., 0%) on the progress bar. As another example, the developer of k9Mail, in our illustrative example (Section 4.2), may decide that the proper action to handle *Network Failed* state is to notify the users and ask them to retry once the network is connected. Hence, the appropriate assertion should check the existence of the notification element. Alternatively, the developer may decide to flag the encrypted emails with a specific icon so that the user knows if an email is encrypted or not. In this design decision, the appropriate assertion should check the existence of the specific icons. While these assertions are simple,

Table 4.1: Subject apps

No.	Application Name	Application Category	# DFMs	Size (LoC)	Test-Suite Size Initial	Test-Suite Size Aug.
1	Alert	Productivity	2	914	12	34
2	Amaze File Manager	Utility	3	107141	68	101
3	AntennaPod	Entertainment	2	192094	51	73
4	K9Mail	Utility	2	218242	95	117
5	LeafPic	Utility	2	144332	16	38
6	AnkiDroid	Educational	2	113547	32	54
7	Authorizer	Utility	2	92990	22	44
8	Open Camera	Utility	2	73128	11	33
9	TimeTable	Educational	2	39371	17	39
10	Ping	Utility	2	1633	13	35
11	Bills	Utility	2	55304	12	34
12	Travel Destinations	Lifestyle	2	762	12	34
13	Alkaa To-Do	Productivity	2	46064	37	59
14	Kredit	Utility	2	414	12	34
15	Calculator	Productivity	2	36884	12	34
16	Movie Stats	Entertainment	2	61646	10	32
17	Mediation Ads	Lifestyle	2	29482	4	26
18	MyNews	Educational	2	41764	4	26
19	Dictionary	Educational	2	32841	16	38
20	JackOfAll	Productivity	2	37958	6	28
21	Income Tracker	Productivity	2	31025	19	41
22	Translator	Utility	2	131401	12	34
23	BigFiles	Utility	2	213679	8	30
24	SuperHero	Lifestyle	2	1784	10	32
25	TOKO Game	Game	2	48491	10	32

they require the developer's judgment and understanding of the app's behavior. That said, DELTADROID can suggest the locations where developers could add assertions.

4.5 Evaluation

To evaluate DELTADROID's effectiveness in validating Android apps' dynamic delivery, we investigate the following research questions:

RQ1: *Effectiveness of Inducing Failed States*. How effective is DELTADROID in augmenting test suites to generate test cases that induce different *Failed* states?

RQ2: *Effectiveness of Revealing Installation Defects*. To what extent does DELTADROID reveal installation defects in real-world Android apps' behavior?

RQ3: *Performance*. What is DELTADROID's runtime efficiency in terms of execution time?

4.5.1 Experimental Setup

To answer these research questions, we selected a set of open-source Android apps from GitHub [11] which are also uploaded to Google Play. Our inclusion criteria select apps that (1) are open-source, (2) implement Android dynamic delivery, (3) have at least two DFMs, and (4) have an initial test suite. Note that dynamic delivery features need the developer to sign an app when deployed onto the Google Play platform which, in turn, requires our subject apps to be open source. To select subject apps that implement dynamic delivery, we searched for "`apply plugin: com.android.dynamic-feature`" statement in the Android projects' build files, i.e., Gradle [44]. Among the search results, we excluded toy or demo apps, which are minimal Android apps with no specific functionalities, mainly developed to learn or examine Android dynamic delivery. At the next step, we excluded apps with only one DFM, as one of the *Failed* states, namely *Active Session Limit Exceeded*, requires subjects to have at least two DFMs.

Our final dataset includes 25 apps with a total of 51 DFMs, shown in Table 4.1. These apps come from various categories, reducing any bias towards a certain type of app. Among these 25 apps, the initial test suite of 15 apps has tests to cover all of the DFM installation requests in the code. Since DELTADROID requires baseline test cases that attempt to install DFMs, we manually extended the test suites of the remaining 10 apps with the total number of 20 test cases that initiate the installation of DFMs. The additional test cases are small, i.e., only 10-22 lines of code per test case, and would be straightforward for a developer to write with minimal effort.

For a thorough evaluation of DELTADROID, we first compare it against a baseline static analysis approach, which can detect failures corresponding to the missing implementation of `OnFailureListener` callbacks using Soot [144]. This approach performs a flow-sensitive analysis to find locations in code where an installation request is initiated. It then checks whether the `OnFailureListener` is implemented. In case the `OnFailureListener` callback is not implemented, the static analysis will report a dynamic delivery failure. The rationale here is that without an `OnFailureListener` callback implementation, the app will fail in any of the *Failed State Contexts*. The issue with the static analysis approach is that it does not execute the `OnFailureListener` callback, and, as a result, it does not provide the developers with accurate information regarding their app's behavior under different *Failed State Contexts.*

We also compared DELTADROID against three alternative testing approaches. The first one is Monkey [39], one of the most widely used automated testing tools for Android in practice, which is shown to outperform many other automated testing tools [85]. Second, we compared DELTADROID against APE [123], the state-of-the-art automated model-based approach for testing Android apps, which dynamically optimizes the model based on the run-time information. APE is shown to outperform similar state-of-the-art Android GUI testing tools (*Sapienz*[158] and *Stoat*[189]) in terms of both testing coverage and the number of detected unique crashes. Third, we compared DELTADROID against App Crawler [50] that tests an Android app by running alongside the app and automatically issuing actions, e.g., tap, swipe, etc., to explore the state-space of the target app. It automatically terminates when there are no more unique actions to perform, the app crashes, or after a designated timeout.

Test augmentation techniques target a specific goal in testing, e.g., achieving higher coverage of the code or reproducing the crashes. Unlike those techniques, DELTADROID uses test augmentation to cover all the failed states in the lifecycle of a DFM installation request by relying on a novel defect model. None of the other test augmentation approaches target the same goal. Hence, they are not suitable candidates for comparison against DELTADROID. That said, we tried to compare DELTADROID with Thor [56], a test suite augmentation technique in the domain of Android, since it creates events that can induce the *Network Error* and *Access Denied* failed states (but not other failed states). However, the current version of Thor is outdated, as also mentioned in their issue repository [53]. We have also contacted the authors who confirmed that Thor is no longer actively maintained and, since it is built on Android 4, it will not support Android apps with dynamic delivery.

In addition, we were unable to compare our results against *TimeMachine* [104], since the current version of it does not support Google Play Services, as reported on its issue repository [54], preventing us from running it on app bundles that use dynamic delivery.

Note that no other testing tool is quite comparable to DELTADROID, as none of those tools generate the necessary system events for testing Android dynamic delivery. Explicitly handling dynamic delivery of Android apps (e.g., DFMs), and representing and accounting for the corresponding defect model and *Failed State Contexts*, are key novel elements of DELTADROID that contribute to its testing effectiveness.

For the purpose of comparison, we gave Monkey and APE an hour to run, similar to prior studies [123, 85]. We also ran App Crawler without any designated timeout to be able to explore all states of the application. To further avoid any bias in favor of DELTADROID, we re-installed the apps on the device once any test by Monkey, APE, and App Crawler completed a DFM installation. That is mainly due to the fact that once a DFM is installed, a *Failed* state cannot possibly be reached. Therefore, alternative approaches had many chances to cover *Failed* states during the hour-long period.

4.5.2 RQ1: Effectiveness of Inducing Failed States

To answer RQ1, we ran DELTADROID over all the subject apps to augment their existing test suites. Table 4.1 shows the size of the initial and augmented test suites for each subject app under the *Test-Suite Size* column. These results confirm that DELTADROID was able to generate a new test case for each *Failed* state for subject apps' DFMs, i.e., the size of the initial test suite is increased by $\#DFM \times 11$ tests.

We executed augmented test suites and ran Monkey, APE, App Crawler, and the baseline static analysis approach on the subject apps. Each app is instrumented (Section 4.4.1); we thus monitored the apps' log statements during test execution to identify whether the tests covered different *Failed* states. If the log record of a test indicates that the installation request entered any of the *Failed* states, we confirm that the test covers the state. We also ran the applications' initial test suite without manually extending them, and they could not test any of the defined failed states, as these states occur only under peculiar contextual settings.

Table 4.2: Testing tools’ effectiveness of covering *Failed* states

	Monkey [39]			APE [123]			App Crawler [50]			Baseline Static Analysis		DELTADROID		
Failed State	# Apps	# DFMs	Avg. T.T.F.[1]	# Apps	# DFMs	# T.T.F.	# Apps	Avg. DFMs	# T.T.F.	# Apps	Avg. DFMs	# Apps	# DFMs	Avg. T.T.F.
N.E.	19	38	611	0	0	-	-	-	-	9	19	25	51	7
I.S.	0	0	-	0	0	-	-	-	-	9	19	25	51	9
A.D.	0	0	-	0	0	-	-	-	-	1	1	1	1	13
I.W.E.S.	0	0	-	0	0	-	1	1	3	9	19	25	51	5
A.S.L.E.	4	5	193	1	1	221	2	2	2	9	19	25	51	6
M.U.	0	0	-	0	0	-	-	-	-	9	19	25	51	4
A.N.A.	0	0	-	0	0	-	-	-	-	9	19	25	51	5
A.S.A.N.F.	0	0	-	0	0	-	-	-	-	9	19	25	51	5
S.N.F.	0	0	-	0	0	-	-	-	-	9	19	25	51	4
A.N.O.	0	0	-	0	0	-	-	-	-	9	19	25	51	4
I.R.	0	0	-	0	0	-	-	-	-	9	19	25	51	6
Avg. Total time per app	3600s			3600s			52s			53s		146s		
% Installed DFMs	100%			100%			100%			N/A[2]		100%		

[1] T.T.F., short for Time To Failed states, indicates the testing time required to reach the failed state.
[2] The baseline static anslysis approach does not initiate DFM installations, since it statically analyzes the apps.

Table 4.2 demonstrates the results of this study. It shows that DELTADROID was able to reach almost all of the *Failed* states for the DFMs in a total of 25 subject apps. The only exception was the *Access Denied Failed* state, which was only reached by the tests in one app. The scenario in which an installation request can enter the *Access Denied Failed* state is quite a rare case that requires the app to initiate the installation from a concurrent thread that can be executed while the app is in the background (see Equation 4.3). Only one of the apps among our subjects, namely *Travel Destinations*, was implemented in such a way, for which DELTADROID was able to cover the *Access Denied Failed* state.

Monkey, APE, and App Crawler were all able to generate tests that installed all the DFMs in the subject apps, as reported in Table 4.2. However, they all performed quite poorly in terms of testing different *Failed* states. Specifically, APE was able to only cover the *Active Sessions Limit Exceeded Failed* state for 1 DFM. App Crawler performed better than APE and covered two *Failed* states, i.e., *Incompatible With Existing Sessions* in 1 DFM of 1 app and *Active Sessions Limit Exceeded* in 2 DFMs of 2 different apps. While Monkey performed better than APE and App Crawler in covering *Failed* states, it was far behind DELTADROID. That is, Monkey was able to reach the *Network Error Failed* state for 38 DFMs of 19 apps and the *Active Sessions Limit Exceeded Failed* state in 5 DFMs of 4 apps, making it cover 2 *Failed* states and 43 DFMs in total.

The baseline static analysis approach was able to inform us about missing the implementation of `OnFailureListener` callbacks in 19 DFMs of 9 apps, indicating the possibility of all types of Failed states for these apps. However, the static analysis approach results in a false negative in case the `OnFailureListener` is implemented, but not in a way to properly handle the failed states. Additionally, static analysis cannot generate tests to help developers induce the failed state contexts.

Overall, Table 4.2 demonstrates that DELTADROID significantly outperforms alternative approaches in testing Android dynamic delivery. Additionally, these results confirm our choice to formulate dynamic delivery testing as a test-suite augmentation technique, since even a pure random testing technique, such as Monkey, can reach program points that install DFMs.

4.5.3 RQ2: Effectiveness of Revealing Installation Defects

To assess the DELTADROID's ability to reveal defective behavior in Android apps, we investigated each app's behavior in our dataset once any of the *Failed* states occur. Proper behavior for an app entering a *Failed* state could notify the users of the situation, explain the reason for failure, give them an instruction, or retry the installation as described in Android documentation [49]. However, in our study, we conservatively consider a defective behavior only when an app crashes (*Crash*) or takes no action to handle the failure, i.e., no change in the behavior of the app (*Unhandled*), such as the illustrative example in Section 4.2. As a result of this study, we found 18 apps (i.e., 72%) with the total of 160 unhandled behaviors upon the occurrence of a *Failed* state. More importantly, we discovered 48 new instances of app crashing in 7 apps (i.e., 28%), which were not detected by their initial test suites. Unlike a typical crash where simply restarting the app is sufficient to return the app to a stable state, in all of these 7 crashing apps, we had to manually uninstall and install the apps to resume their regular functionalities. We believe this is because DFM-induced crashes occur during the installation of new modules that, if not completed properly, corrupt the app bundle permanently. DELTADROID's oracle was able to successfully determine the crashes, handled, and unhandled behaviors in 23 apps in our dataset (i.e., 92%). For the remaining 2 apps, the oracle resulted in false positives, i.e., incorrectly identified an unhandled behavior as handled since a UI element appeared in the screen after the failure, but not for the purpose of handling it.

Table 4.3 describes the apps' detailed behavior in our dataset under different *Failed* states. Notations "UH" (colored in yellow) and "CR" (colored in red), respectively, imply unhandled behaviors and crashes under the *Failed States* columns. The checkmark notation (colored in green) indicates that the app handles the corresponding *Failed* state by taking an extra action. All of the 6 apps in our dataset that handle at least one *Failed* state only display an error message without providing any instructions to the user. Excluding the rare scenario of *Access Denied Failed* state, our evaluation results indicate that for different *Failed* states, the unhandled behavior ranges from 60% to 64%, and crashes range from about 16% to 24% of the apps. These results emphasize the importance of testing Android dynamic delivery and the consequences of disregarding it.

Table 4.3: Installation defects revealed by DELTADROID

No.	Application Name	Failed States										
		N.E.	I.S.	A.D.	I.W.E.S.	A.S.L.E.	M.U.	A.N.A.	A.S.A.N.F.	S.N.F.	A.N.O.	I.R.
1	Alert	UH	UH	-	UH	CR	UH	UH	UH	UH	UH	UH
2	Amaze File Manager	UH	UH	-	UH	UH	UH	UH	UH	UH	UH	UH
3	AntennaPod	✓	✓	-	✓	✓	✓	✓	✓	✓	✓	✓
4	K9Mail	UH	CR	-	CR	CR	CR	CR	CR	CR	CR	CR
5	LeafPic	UH	UH	-	UH	UH	UH	UH	UH	UH	UH	UH
6	AnkiDroid	UH	UH	-	UH	UH	UH	UH	UH	UH	UH	UH
7	Authorizer	✓	✓	-	✓	✓	✓	✓	✓	✓	✓	✓
8	Open Camera	UH	UH	-	UH	UH	UH	UH	UH	UH	UH	UH
9	TimeTable	UH	UH	-	UH	UH	UH	UH	UH	UH	UH	UH
10	Ping	UH	UH	-	CR	CR	UH	✓	✓	✓	✓	✓
11	Bills[1]	UH	UH	-	UH	UH	UH	UH	UH	UH	UH	UH
12	Travel Destinations	✓	✓	✓	CR	✓	✓	CR	CR	CR	CR	CR
13	Alkaa To-Do	UH	UH	-	UH	UH	UH	UH	UH	UH	UH	UH
14	Kredit	✓	✓	-	UH	UH	✓	UH	UH	UH	UH	UH
15	Calculator	CR	CR	-	CR	CR	CR	CR	CR	CR	CR	CR
16	Movie Stats	UH	UH	-	UH	UH	UH	UH	UH	UH	UH	UH
17	Mediation Ads	UH	UH	-	UH	UH	UH	UH	UH	UH	UH	UH
18	MyNews	UH	UH	-	UH	UH	UH	UH	UH	UH	UH	UH
19	Dictionary	CR	CR	-	CR	CR	CR	CR	CR	CR	CR	CR
20	JackOfAll	UH	UH	-	UH	UH	UH	UH	UH	UH	UH	UH
21	Income Tracker	UH	UH	-	UH	UH	UH	UH	UH	UH	UH	UH
22	Translator	CR	CR	-	CR	CR	CR	CR	CR	CR	CR	CR
23	BigFiles	UH	UH	-	UH	UH	UH	UH	UH	UH	UH	UH
24	SuperHero	✓	✓	-	✓	✓	✓	✓	✓	✓	✓	✓
25	TOKO Game	UH	UH	-	UH	UH	UH	UH	UH	UH	UH	UH
% of apps not handling the failed state		64%	64%	0%	64%	60%	64%	64%	64%	64%	64%	64%
% of apps crashing in the failed state		16%	16%	0%	24%	24%	16%	20%	20%	20%	20%	20%

[1] Bills app freezes in every Failed state in which the user cannot have any further interaction unless the app is restarted or forced to quit.

One important outcome of these results is that a mere static analysis is not effective for validating dynamic delivery. Specifically, static analysis techniques, e.g., the baseline static analysis approach, at best, can only determine dynamic delivery failures corresponding to missing implementation of "OnFailureListener" callback. Such failures constitute 70 out of 160 unhandled behaviors and 10 out of 48 crashes in our results. This is because dynamic delivery failures depend on contextual settings, e.g., regarding network connection, permissions, device storage, etc., that require app execution and dynamic analysis.

Finally, we reported all the discovered issues to the subject apps developers. As of this submission date, 21 crashes and 54 unhandled behaviors are confirmed by the developers. Developers' responses also confirm the wide-reaching importance of the problem, the difficulty of testing Android dynamic delivery, and their interest in using our technique. For example, one developer indicated that they had not considered testing the installation failure scenarios before our report. The developer said, "As a developer, we try to cover all the failure scenarios, and testing them is not straightforward. Therefore, a testing tool that could detect all the loopholes for dynamic delivery implementation would be very helpful to us." Another developer noted that "The Android documentation is not very clear, and it is indeed difficult to test traditionally, or it requires a lot of code. I think a framework or utility to check the behavior of Android apps when module installation fails could be very useful".

4.5.4 RQ3: Performance

To investigate the performance characteristics of DELTADROID, we first assess the performance of its two main components, i.e., *Test-Suite Analysis* and *Test-Suite Augmentation*. We next compare the efficiency of DELTADROID to alternative approaches. We ran all of the tools on a Pixel 3a mobile device with Android 10 and an Android Emulator with Android 9 running on a MacBook Pro 2013 (2.3 GHz Intel Core i7, 16 GB, MacOS 10.14), depending on the Android versions of the apps.

To assess the performance characteristics of DELTADROID, we measured the execution time corresponding to each of its components, as shown in Table 4.4. On average, it takes about 69 seconds for DELTADROID to perform all of the required analysis on the initial test suite and 8 seconds to generate new test cases. The overall average execution time of 77 seconds confirms the scalability of DELTADROID for validating Android apps' dynamic delivery.

Table 4.4: Execution time of DELTADROID

Phase	Avg. Execution Time (s)
Test-Suite Analysis	68.94
Test-Suite Augmentation	8.25
Total	77.19

To investigate the efficiency of DELTADROID compared to alternative approaches, listed in Table 4.2, we measured the number of discovered dynamic delivery faults over a specific amount of time for each approach. To that end, we considered the maximum execution time of all approaches, i.e., an hour for Monkey and APE. Figure 4.8a shows the resulting plots with each approach in a different color. Then, to better compare the effectiveness of all candidate approaches, we measured the corresponding area-under-the-curve (AUC) for each plot. Figure 4.8b shows the amount of AUC of the mentioned plots for each approach. The results indicate that DELTADROID's efficiency in revealing dynamic delivery faults is exceedingly superior to alternative approaches.

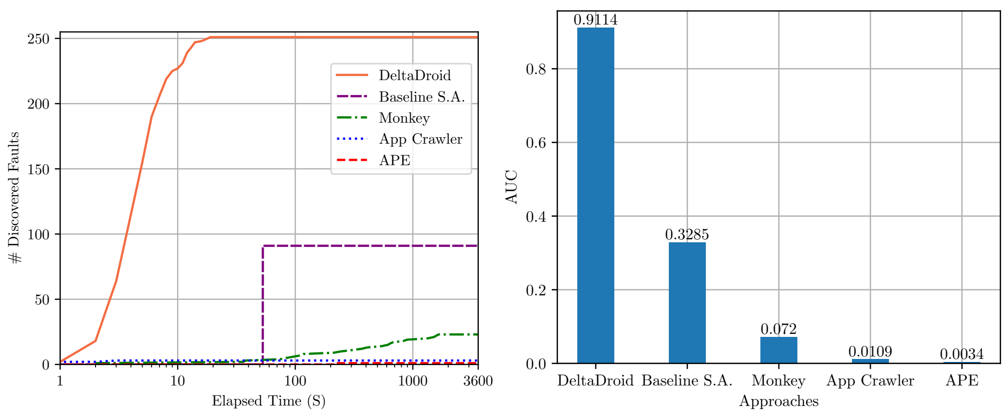


Figure 4.8: The results of comparing alternative approaches in revealing dynamic delivery faults over time. (a)The number of discovered faults over the elapsed time for all approaches listed in Table 4.2. (b)The Area-Under-the-Curve (AUC) of the plots in Figure 4.8a.

Furthermore to compare DELTADROID's efficiency with alternative approaches, we measured the average time to cover a *Failed* state for the first time in each testing approach, i.e., Monkey, APE, and App Crawler. This metric does not apply to the baseline static analysis approach since it statically analyzes the apps' source code, reports the final results, and terminates the execution. Table 4.2 shows the results of this study (under columns *Avg. T.T.F.* which is short for Time to Failed state). DELTADROID takes about 6 seconds on average to reach a *Failed* state, with a minimum of 4 and a maximum of 13 seconds. The corresponding numbers for Monkey, APE, and App Crawler are 402, 221, and 2.5 seconds on average, respectively. Although App Crawler reaches the failed states exceedingly fast compared to DELTADROID, it performs quite poorly in terms of covering different failed states.

In addition, we calculated the average testing time of each technique and the total execution time for the baseline static analysis approach among all subjects. The results show that the test cases generated by DELTADROID took a total time of 146 seconds, on average, to test each app. On the other hand, Monkey and APE were allowed to run for an hour on each app. App Crawler completed testing each app after 52 seconds, on average. The events DELTADROID injects, e.g., connecting and disconnecting Wi-Fi for the Network Error failed state, or filling the device's remaining storage for the Insufficient Storage failed state, take more time compared to GUI events, e.g., clicking buttons. Therefore, DELTADROID is slightly slower compared to App Crawler. The baseline static analysis approach also took a total time of 53 seconds, on average, to execute on each app. Although the App Crawler and the baseline static analysis approach execute more quickly than other techniques, these faster techniques are substantially inferior to DELTADROID in terms of covering different failed states and revealing dynamic delivery defects.

4.6 Threats to Validity

The main threat to internal validity is that DELTADROID, indeed as a test augmentation technique, requires initial tests that reach program points that install DFMs. In case the test suite does not include test cases initiating the installation of DFMs, DELTADROID reports no baseline tests to augment. Therefore, DELTADROID would not be able to generate new test cases to test Android dynamic delivery failures. However, these test cases are easily obtained. According to our investigation, even a pure random testing technique such as Monkey [39] was able to create such tests for all apps, as reported in Table 4.2. For the evaluation of DELTADROID, we have also manually extended the existing test suites of a subset of subject apps with test cases that attempt to install their DFMs, and reported that these test cases were small, i.e., only 10-22 lines of code per test case, and would be straightforward for a developer to write.

One of the main threats to external validity is the selection of subject Android apps in our evaluation. To mitigate this threat, we selected open-source Android apps that (1) implement Android dynamic delivery, (2) have at least two DFMs, and (3) have an initial test suite among the hundreds of applications on GitHub [11], one of the largest and most widely used open-source repositories online. Another threat to external validity is whether the types of dynamic delivery failed state contexts, defined in our defect model, accurately describe the existing dynamic delivery failures. To alleviate this threat, we explored multiple resources, including the Android documentation, issue repositories, discussion forums, and open-source Android apps' source code, to understand the root causes of dynamic delivery failures and identify contextual factors to induce them. Note that to construct a defect model for the purpose of test generation, we only need one situation in which the desired failed state is induced.

4.7 Related Work

This section provides an overview of the prior work on test-suite augmentation and test-input generation for mobile apps.

Test-Suite Augmentation: Software evolves, so should the test suites to ensure validation of the evolving software. The purpose of test-suite evolution is two-fold [173, 221]. First, it can be used to repair available test cases for corrective regression testing [163, 127, 120, 217]. Second, developers can create additional tests for either progressive regression testing [65, 181, 216, 215, 214, 224, 178, 213, 137, 203, 222, 98, 223, 95] or to fulfill additional testing criteria, e.g., achieve higher coverage, find more crashes, or perform non-functional testing [164, 218, 174, 101, 60, 139, 172, 56]. Reuse of the available test suites to generate additional tests is known as *test augmentation*.

The closely related work to DELTADROID are [56, 164, 218, 174]. Adamsen et al. proposed a technique that augments the existing test suites and systematically exposes them to adverse conditions where certain unexpected events may interfere with the test execution. They realized their technique in a tool, Thor, working on Android. Milani Fard et al. [164] proposed a technique that leverages existing test suites to automate the generation of tests for web applications. Specifically, they mine the human knowledge from the existing test suite to generate additional tests that explore uncovered parts of the program. Xuan et al. [218] proposed a technique to reproduce crashes by leveraging the existing test suites, instead of automatic generation of new tests. Pradhan et al. [174] proposed a search-based technique to obtain the program dependence graph from the existing test suite to generate new tests to cover untested program configurations.

Although Thor [56] is similar in terms of using test augmentation techniques to provide contextual factors in running existing test suites, the events it injects into the test cases are substantially different from DELTADROID's GUI and system events. It only generates events that might simulate two failed state contexts, namely *Network Error* and *Access Denied*. Therefore, it cannot be used to comprehensively test Android dynamic delivery failures. Furthermore, unlike [164, 174] that tries to achieve a higher *coverage of the code* by augmenting the existing test suites, the goal of DELTADROID is to cover all the *Failed* states in the lifecycle of a DFM installation request through test augmentation. Moreover, While [218] relies on the information from a stack trace of a crash report to identify target tests and augments the existing test suite using random mutation operators, DELTADROID relies on a novel defect model that formally defines failure contexts to generate effective tests.

Android Testing: Android test generation techniques mainly focus on either fuzzing to generate inputs or exercise an Android app through its GUI. The majority of recent techniques [189, 123, 92] rely on a GUI model, usually constructed dynamically and non-systematically, leading to unexplored program states. Sapienz [158], EvoDroid [152], and time-travel testing [104] employ an evolutionary algorithm. ACTEve [63], and Collider [136] utilize symbolic execution. CrawlDroid [78] introduces a feedback-based exploration strategy to effectively explore different states of Android apps. AimDroid [122] discovers unexplored activities with a reinforcement learning guided random algorithm. AppFlow [131] leverages machine learning to automatically recognize common screens and widgets and generate tests accordingly. MonkeyLab [149] mines app executions to generate GUI-based scenarios. Dynodroid [151] and Monkey [39] generate test inputs using random input values. Another group of techniques focuses on testing for specific defects [128, 225, 134, 179]. Mao et al. introduce a different approach [159] that generates replicable test scripts from crowd-based testing by collecting and analyzing test inputs from a crowd call to non-technical users with no specific software testing expertise or experience. None of these techniques can be used to properly validate the dynamic delivery in Android apps, as they cannot generate meaningful test inputs to induce the contextual settings for the manifestation of *Failed* states.

Similar to DFM-induced defects, energy defects occur under peculiar contextual settings. Jabbarvand et al. [134] proposed a search-based technique that leverages contextual models to generate both system and GUI test inputs. Unlike [134], which searches for contextual settings that manifest energy defects using meta-heuristics, DELTADROID use a formal model of such contexts to generate tests for all of them.

4.8 Discussion

In this chapter, we formally defined a novel defect model representing the conditions (contexts) under which installation of a Dynamic Feature Module (DFM) in an Android app could fail. Utilizing the defect model, we introduced DELTADROID, a test-suite augmentation approach for testing dynamic delivery in Android apps. DELTADROID leverages static and dynamic analyses to detect the test cases that initiate the installation of DFMs in apps. DELTADROID then modifies these tests to create new tests that augment the initial test suite to effectively create the conditions for reaching all of the *Failed* states. Our experimental results corroborate the effectiveness of DELTADROID in inducing *Failed* states and detecting a significant number of installation defects among real-world Android apps. We have made DELTADROID publicly available [51]. In our future work, we aim to extend DELTADROID to test dynamic delivery in other platforms, e.g., Java Platform Module System [7].

Chapter 5

Run-time Automatic Architectural Inconsistency Resolution in Java

Many mainstream programming languages lack extensive support for architectural constructs, such as software components, which limits software developers in employing many benefits of architecture-based development. To address this issue, Java, one of the most popular and widely-used programming languages, has introduced the Java Platform Module System (JPMS) in its 9th and subsequent versions. JPMS provides the notion of architectural constructs, i.e., software components, as an encapsulation of modules that helps developers construct and maintain large applications efficiently—as well as improving the encapsulation, security, and maintainability of Java applications in general and the JDK itself. However, ensuring that module declarations reflect the actual usage of modules in an application remains a challenge that results in developers mistakenly introducing inconsistent module dependencies at both build- and run-time. Chapter 3 describes JPMS properties and architectural notions in-depth and defines a defect model consisting of eight inconsistent modular dependencies that may arise in Java applications. Based on the defect model, in chapter 3.3, we presented DARCY, a framework that leverages the defect model and static analysis techniques to automatically detect and repair the specified inconsistent dependencies within Java applications, specifically at build-time. In this chapter, we extend DARCY and present DARCY$^+$ that supports the automatic resolution of architectural inconsistencies that may arise at the run-time, as well as the build-time, of Java applications.

The results of our experiments, conducted over 52 open-source Java 9+ applications, indicate that architectural inconsistencies are widespread and demonstrate the DARCY$^+$'s effectiveness in the automated resolution of these inconsistencies at run-time.

5.1 Introduction

Every software system has an architecture comprising the principal design decisions employed in the system's construction [taylor2008], which is not usually explicitly documented, e.g., in the form of UML models. Furthermore, the architecture of a system is often conceptualized in terms of high-level constructs, such as software components, connectors, and their interfaces. However, programming languages used to implement the software systems provide low-level constructs, such as classes, methods, and variables. Therefore, in many software systems, the architecture as intended or even documented, known as the prescriptive architecture, does not match the architecture reflected in the system's implementation, known as the descriptive architecture, and it is a non-trivial task to map one to the other. Hence, ensuring the conformance between the prescriptive and descriptive architecture has been a significant challenge in the software engineering literature [taylor2008].

As also discussed in chapter 3, inconsistencies between the prescriptive and descriptive architectures are highly critical and need to be resolved, as a system's architecture determines its key properties. Therefore, the software engineering research community has introduced several approaches to minimize such inconsistencies, including frameworks or programming languages that provide the realization of architectural constructs in the implementation of software applications [cite several]. However, many mainstream programming languages have lacked extensive support for architecture-based development. This all changed with the introduction of Java Platform Module Systems (JPMS) in Java 9 and subsequent versions[1]. JPMS incorporates the notion of modules, which represents software components, as a Java programming language construct. Hence, developers can explicitly specify the software system's components, their interfaces, and their detailed dependencies amongst each other in a file called `module-info.java`, which forms the system's prescriptive architecture.

[1]We may refer to Java 9 and subsequent versions as Java 9+ in this chapter.

In chapter 3.3 we discussed the inconsistencies between the prescriptive, i.e., dependencies specified in the `module-info.java` file, and the descriptive architecture, i.e., the system's implemented dependencies at build-time. Then, we presented DARCY, an automated framework that leverages a defect model consisting of 8 types of architectural inconsistencies we formally defined and static analyses to automatically resolve the identified inconsistencies within Java applications statically at build-time (refer to Chapter 3.4.) Similarly, in case of any changes to the specified dependencies of modules at run-time, Java cannot ensure whether those changes are consistent with the applications' implemented dependencies. In this chapter, first, we extend our previous work and present DARCY^{+} to detect and repair architectural inconsistencies not only statically at build-time but also dynamically at run-time that may arise due to dynamic changes occurring to software systems at run-time. Second, we designed and reported on new experiments to evaluate DARCY^{+}'s ability to resolve architectural inconsistencies at run-time and improve architectural metrics followed by the resolution.

The results of our experiments, conducted over 52 open-source Java 9+ applications, indicate that architectural inconsistencies are widespread and demonstrate the benefits of DARCY^{+} in automated detection and repair of these inconsistencies at run-time. By automatically fixing these inconsistencies, DARCY^{+} was able to measurably improve various attributes of the subject applications' architectures at run-time by reducing the attack surface of applications, improving their encapsulation, and producing deployable applications that consume less memory.

The remainder of this chapter is organized as follows. Section 5.2 reviews the Java module system and its design goals. Section 5.3 provides a brief overview of the formally specified defect model, introduced in Chapter 4.3, consisting of the architectural inconsistencies in the context of Java 9+. Section 5.4 provides details of DARCY^{+} and its implementation. Section 5.5 presents the experimental evaluation of the research. Section 5.6 includes the threats to validity of our approach. The chapter concludes with an outline of related research and a discussion.

5.2 Overview of Java Platform Module System

Chapter 3.2 introduces the module system in Java 9+, called Java Platform Module System (JPMS), and provides an overview of JPMS's goals and the architectural risks that arise from its misuse. More specifically, it discusses the details of modules in Java 9+ —including module declarations, module directives, and their behavior at run-time.

Regarding Java application's run-time behavior, JPMS also allows developers to modify Java applications at run-time. More specifically, developers can dynamically load new modules into a running Java application or unload them [153]. Depending on different use cases, Java modules can be dynamically loaded or unloaded at run-time [153], altering the system's prescriptive and descriptive architecture.

5.3 Defect Model: Inconsistent Module Dependencies

As described earlier, architectural inconsistencies may arise when using modules, based on the module directives described in the previous section and Chapter 3.2. These inconsistencies occur in modules' specifications at both build-time and run-time. Chapter 3.3 describes the architectural inconsistencies in details. This chapter focuses on the architectural inconsistencies that may arise during software systems' run-time.

Similar to build-time, insufficiently specified dependencies (e.g., a module that attempts to use a package it does not have a required directive for) are already checked by the Java platform but cause a run-time exception. However, excess dependencies, where a module either (1) exposes more of its internals than are used or (2) requires internals of other modules that it never uses, are not handled by Java. These inconsistencies can affect various architectural attributes, including encapsulation and maintenance, software bloat and scalability, and security.

To achieve a systematic and comprehensive coverage of all types of inconsistent module dependencies, we studied all potential inconsistencies resulting from developers' misuse of each type of module directive. The result of this study led to a defect model specifying eight types of inconsistent dependencies that may arise when using JPMS, which is described in Chapter 3.3 in detail.

The architecture of software systems can change during their run-time, e.g., introducing new dependencies by loading new modules. The new modules' specified dependencies, i.e., its prescriptive architecture, might not reflect the implemented dependencies in the module, i.e., its descriptive architecture. Since Java platform does not provide any mechanism to ensure the consistency between the specified and implemented dependencies, all types of inconsistent module dependencies can occur at run-time as well as the build-time. Therefore, in the remainder of this chapter, we leverage the defect model explained in chapter 3.3 to develop DARCY$^+$ that resolves modular run-time inconsistencies.

5.4 DARCY$^+$: Framework Overview

This section describes how we designed and developed DARCY$^+$ to automatically resolve architectural inconsistencies at both build- and run-time, leveraging the defect model of inconsistent module dependencies and the components in our previous approach, presented in Chapter 3.4. Figure 5.1 depicts a high-level overview of DARCY$^+$ comprised of two phases, Detection, and Repair. Each DARCY$^+$'s component works at either build-time, run-time, or both, color-coded in Figure 2. DARCY$^+$ is implemented in Java and Python.

5.4.1 Detection

The detection phase takes a Java application as input and identifies any instance of the eight inconsistent dependencies described in Chapter 3.3. In this section, we briefly review the DARCY$^+$'s components that were introduced in our previous work, explained in Chapter 3.4. Then, we describe the additional components that were added as a part of DARCY$^+$'s run-time extension in detail.

To identify the implemented dependencies of an input Java application, DARCY$^+$ relies on static analysis, represented as *Package Dependency Analysis* in Figure 2, for which we leveraged Classycle [106]. More specifically, we ran Classycle to collect the information about implemented dependencies in the source code

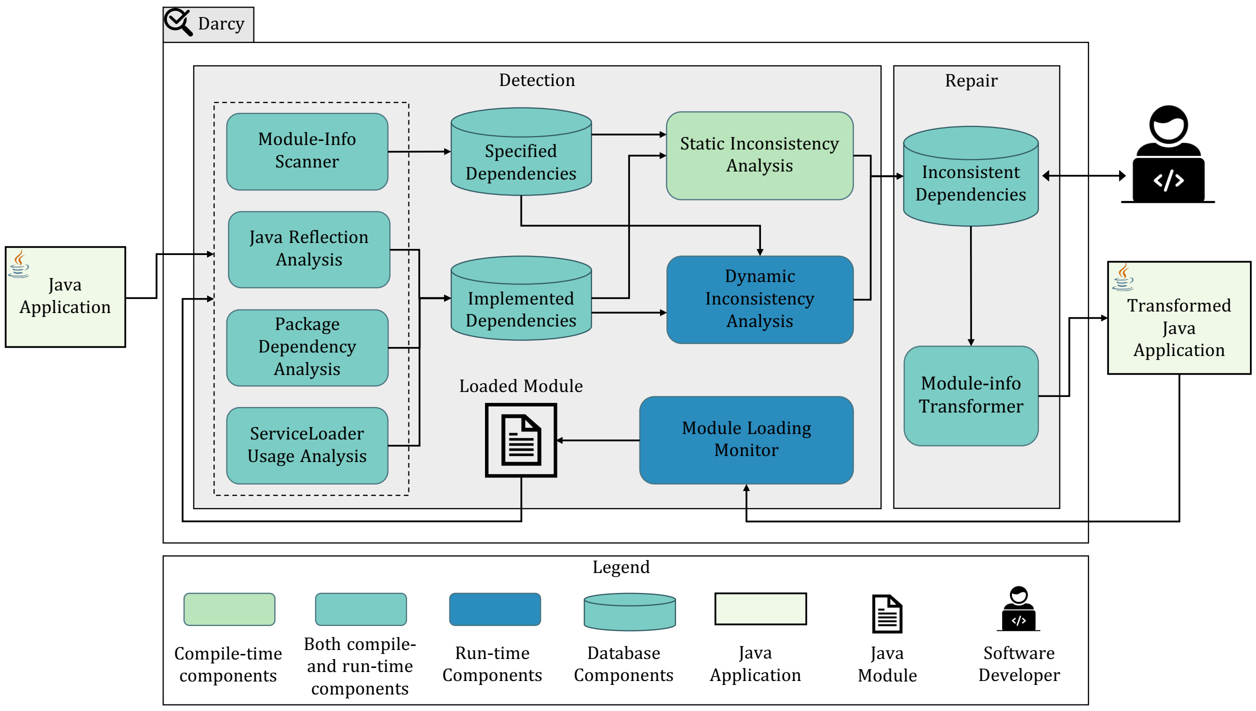


Figure 5.1: A high-level overview of DARCY$^+$

of the input application, which provides a complete report of all dependencies in the source code of a Java application at both the class and package levels. Since the dependencies defined in modules are at the package level, we only extracted the dependencies among packages. The results of *Package Dependency Analysis* are stored in *Implemented Dependencies*, which is a database component.

More precisely, the information about implemented dependencies in the source code of the input application is collected by running Classycle, which provides a complete report of all dependencies in the source code of a Java application at both the class and package levels. We only need the extracted dependencies among packages since the dependencies defined in modules are at the package level. *Package Dependency Analysis*'s results are stored in *Implemented Dependencies*, which is a database component.

Module-Info Scanner examines all `module-info.java` files within the input Java application and extracts all specified dependencies which are defined at the package level. The collected information of specified dependencies is stored in another database component, *Specified Dependencies*.

Java Reflection Analysis leverages a custom static analysis [113], which we have implemented using the Soot framework [200], to identify the usage of reflection in the input application. *Java Reflection Analysis* extracts reflective invocations that occur in cases where non-constant strings, or inputs, are used as target methods of a reflective call. Recall that a reflective invocation of a method, for both constructor and non-constructor methods, occurs in three stages: (1) class procurement (i.e., a class with the method of interest is obtained), (2) method procurement (i.e., the method of interest to be invoked is identified), and (3) the method of interest is actually invoked. *Java Reflection Analysis* attempts to identify information at each stage. Finally, the traces of any actual usage of reflection in the Java application is then stored in *Implemented Dependencies*.

ServiceLoader Usage Analysis extracts the implemented dependencies of type *uses* leveraging a custom static analysis using the Soot framework to identify usage of **`java.util.ServiceLoader`** in the input application. Recall that an application obtains a service loader for a given service by using the **`ServiceLoader`** API. A service loader can locate and instantiate providers of the given service [18]. Finally, the traces of any actual usage of services is then stored in *Implemented Dependencies*.

All the above-mentioned components are used at both build-time, for the whole application, and run-time, for the loaded modules, to collect all specified and implemented dependencies of Java modules to further assist the detection of architectural inconsistencies. Refer to Chapter 3.4 for an in-depth description of the mentioned components.

Detection at Build-time

Static Inconsistency Analysis's main goal is to identify all types of inconsistency scenarios described in Section 3.3 at build-time. For each directive in a `module-info.java` file, *Static Inconsistency Analysis* explores implemented and specified dependencies, stored in their respective database components, to identify any occurrence of an inconsistent dependency defined in Section 3.3. If a matching instance is found, *Static Inconsistency Analysis* reports the identified architectural inconsistency, the module affected, and the specific directive involved. The component then stores the identified inconsistencies in *Inconsistent Dependencies*, which are then used in the repair phase.

Detection at Run-time

Recall that a Java application can dynamically load a module while running. Hence, its architecture and module dependencies can change at run-time. Subsequently, the loaded module can contain architectural inconsistencies, which will be introduced to the Java application at run-time. The *Module Loading Monitor* and *Dynamic Inconsistency Analysis* components have been added to our previous work, explained in Chapter 3.4, in developing DARCY$^+$ to automatically detect architectural inconsistencies at run-time.

The *Module Loading Monitor* monitors the running Java application for any dynamic changes to its dependencies, i.e., dynamically loading a new module. In case a new module is dynamically loaded to the system, as mentioned in Section 5.2, the *Module Loading Monitor* notifies and passes the loaded module's information to *Module-info Scanner*, *Java Reflection Analysis*, *Package Dependency Analysis*, and *ServiceLoader Usage Analysis* components. Unlike the detection phase at build-time, the components mentioned above only analyze the loaded module and collect the additional implemented and specified dependencies into their corresponding database components. Afterward, *Dynamic Inconsistency Analysis* component analyzes the change in the architecture using the previously-stored dependencies at build-time. It then identifies all types of inconsistency scenarios described in Section 3.3 that arise by the loaded module at run-time. The *Dynamic Inconsistency Analysis* analyzes the added dependencies by the loaded module at run-time, whereas *Static Inconsistency Analysis* analyzes the whole application at build-time. The inconsistencies detected by *Dynamic Inconsistency Analysis* will be added to the *Inconsistent Dependencies* to be used later in the repair phase.

5.4.2 Repair

To repair inconsistent dependencies, the repair phase transforms modules with detected inconsistent dependencies both statically at build-time and dynamically at run-time. *Module-Info Transformer* is responsible for transforming module specifications to match their implemented dependencies.

Module-Info Transformer is responsible for repairing inconsistent dependencies in a module by deleting or modifying the explicit dependencies defined in the `module-info` files. All module inconsistencies detected in the detection phase are all unnecessarily defined dependencies in `module-info.java` files.

Therefore, *Module-Info Transformer* omits those dependencies specified in the `module-info` files that cause inconsistencies.

The *Inconsistent Dependencies* database includes all the required details regarding the detected inconsistent module dependencies, e.g., the modules and packages involved and the type of the inconsistent dependency. Subsequently, *Module-Info Transformer* collects the related records of inconsistent dependencies defined in each module and modifies the affected lines in their corresponding `module-info.java` files using ANTLR [10]. More specifically, *Module-Info Transformer* uses ANTLR to construct a customized parser based on Java 9+ grammar and modifies it to check the records of inconsistent dependencies found in the detection phase. Upon finding the matched lines in `modue-info.java` files, *Module-Info Transformer* either deletes the entire statement or modifies a part of it, e.g., in case of *Requires Transitive* inconsistency. Furthermore, in the case of `open` module inconsistency (Equation 3.8 in Section 3.3), *Module-Info Transformer* removes the `open` modifier from the module declaration and adds an `opens to` statement for each package that needs to be opened to another module.

Since DARCY$^+$ can not be aware of the intentions of developers and architects, it may repair the specified dependencies with which DARCY$^+$'s users may disagree. For instance, in the case of third-party libraries, the developers intentionally export some packages for further needs or even allow other modules to reflectively access the internals of some classes and packages. In these situations, DARCY$^+$ warns the developers and architects about potential threats caused by architectural inconsistencies in their Java application and allows them to override DARCY$^+$ prior to the application of repairs.

Overall, after the repair phase at build-time, the transformed Java application starts running and will be monitored by *Module Loading Monitor*. In the case of dynamically loading a module at run-time, the detection and repair phase will be repeated for the loaded module.

5.5 Evaluation

To assess the effectiveness of DARCY$^+$, we study the following research questions:

RQ1: How accurate is DARCY$^+$ at resolving inconsistent, architectural dependencies dynamically at run-time?

RQ2: To what extent does DARCY$^+$ reduce the attack surface of Java modules at run-time?

RQ3: To what extent does DARCY$^+$ enhance encapsulation of Java modules at run-time?

RQ4: To what extent does DARCY$^+$ reduce the memory consumption at run-time?

RQ5: What is DARCY$^+$'s run-time efficiency in terms of execution time?

To answer these research questions, we selected a set of Java applications from GitHub [11], a large and widely used open-source repository of software projects, all of which are implemented in Java 9+.

We used a Python script leveraging Beautiful Soup [1], i.e., a web scraping library and GitHub's API, to crawl Github, followed by filtering GitHub repositories. Chapter 3.5 explains the crawling and filtering process in more detail. Finally, our evaluation dataset includes 52 Java 9+ applications. Recall that DARCY$^+$ detected 567 instances of architectural inconsistencies in 38 applications at build-time (refer to Chapter 3.5). Table 3.2 includes these 38 subject applications, their number of modules, packages, and directives.

5.5.1 RQ1: Inconsistency resolution at Run-time

Java applications can dynamically load new modules while running. Hence their architecture changes, and the new modules can introduce further architectural inconsistencies at run-time.

Table 5.1: Identified Inconsistencies and Robustness at Run-time

Application	The Loaded Module			# Runtime	Inconsistencies Types							% Correct	# Runtime
Name	Name	# Packages	# Directives	Incons.	R	R.J.	R.T	E	P	U	O	Incons.	Disruptions
Auto-sort	Unification Service	1	3	1	-	-	-	1	-	-	-	100%	0
Ballerina-lang	Shell	12	19	9	-	-	-	9	-	-	-	100%	0
Blynk-Server	Core	66	62	26	-	-	-	26	-	-	-	100%	0
BunnyHop	BunnyHop	20	23	17	-	1	2	11	3	-	-	100%	0
Codersonbeer-app	Data Model	1	3	2	-	-	-	1	1	-	-	100%	0
Constantin	Database	2	3	2	-	-	1	1	-	-	-	100%	0
Eclipse Jetty	Unixsocket Server	1	7	5	-	-	4	1	-	-	-	100%	0
Java-9-lab	Greeter	1	4	1	-	-	-	1	-	-	-	100%	0
Java-9-modularity	Analysis Service	1	3	1	-	-	1	-	-	-	-	100%	0
Java-Bookstore	Logging	1	2	1	-	-	-	-	-	1	-	100%	0
Java-SPI	API	1	3	2	-	-	1	1	-	-	-	100%	0
Java9-demo	Bank Impl	1	4	1	-	-	-	1	-	-	-	100%	0
Java9-junit	NFO	2	5	1	-	-	-	1	-	-	-	100%	0
Java9-labs	CMS	3	4	3	-	-	-	3	-	-	-	100%	0
java9-modules	Main	1	3	1	-	-	-	1	-	-	-	100%	0
Java9-TLB-modules	Poetry	1	4	1	-	-	-	1	-	-	-	100%	0
JavaUtils	JavaFX	2	7	6	-	-	4	2	-	-	-	100%	0
Jigsaw-resources	App	1	3	1	-	-	-	1	-	-	-	100%	0
Jigsaw-tst	Astro	1	2	1	-	-	-	1	-	-	-	100%	0
Jwtgen	Generator	6	11	1	-	-	-	1	-	-	-	100%	0
Logback	Core	38	33	4	-	-	3	1	-	-	-	100%	0
Meetup	Belarusian Lang	1	4	2	-	-	-	1	-	1	-	100%	0
Music-UI	UI	1	6	2	-	-	-	1	1	-	-	100%	0
Number-to-text	API	3	4	3	-	-	-	3	-	-	-	100%	0
Practical-Security	Banking Server	5	8	2	-	1	-	1	-	-	-	100%	0
Quasar	Core	21	33	16	-	-	-	16	-	-	-	100%	0
QuestDB	IO	67	57	34	-	-	1	33	-	-	-	100%	0
Rahmnathan-utils	Video Converter	4	6	3	-	-	-	3	-	-	-	100%	0
RecolInline-Server	SQL	3	13	7	3	-	1	3	-	-	-	100%	0
Rhizomatic-IO	IO Inject	4	9	2	-	-	-	1	-	-	1	100%	0
Sense-nine	Client	4	10	5	-	1	-	1	3	-	-	100%	0
Sirius	Frontend	1	3	2	-	-	1	1	-	-	-	100%	0
Spring-mvn-java9	Spring MVC	1	7	3	2	-	-	1	-	-	-	100%	0
Springuni-java9	Chat Model	1	2	1	-	-	-	1	-	-	-	100%	0
Tascalate-Javaflow	Agent Proxy	8	13	5	-	-	4	1	-	-	-	100%	0
The-Message	Base	10	11	1	-	-	-	1	-	-	-	100%	0
TRPZ	GUI	1	9	2	-	-	-	1	1	-	-	100%	0
Vstreamer	Player	1	4	3	1	-	-	1	-	-	1	100%	0

To answer RQ1, we simulated the above-mentioned run-time behavior and assessed DARCY$^+$'s capabilities of dynamically detecting and repairing architectural inconsistencies that may arise at run-time.

To simulate this scenario, we randomly selected a module in each dataset application and ran the application without that single module. While DARCY$^+$ is monitoring the application during its run-time, we added the previously selected module as a new module loaded into the system. Then, we evaluated whether DARCY$^+$ can accurately detect inconsistent dependencies of the recently added module at run-time. For this purpose, we manually checked whether the inconsistent dependencies found by DARCY$^+$ match the actual inconsistencies existing in the added module. Table 5.1 describes the results of this experiment, including the number of packages and directives in the added module, as well as the detected run-time inconsistencies separated by their types. As shown in Table 5.1, all inconsistent dependencies found by DARCY$^+$ at run-time are correct.

To evaluate DARCY$^+$'s ability to correctly repair this type of run-time inconsistencies, we ran the repair phase of DARCY$^+$ for each loaded module in our experiment. We manually examined each repair, and all were correct. We further evaluated whether the transformed module can run successfully without any unexpected run-time behavior caused by DARCY$^+$'s repairs. As described in Table 5.1, DARCY$^+$ was able to repair the detected run-time inconsistencies without introducing any disruptions to the applications' run-time, i.e., run-time exceptions. Similar to RQ2, we also ran the detection phase after the repair actions, and the results showed zero inconsistencies within the transformed Java applications.

5.5.2 RQ2: Security

Similar to Chapter 3.5.3, we consider the *attack surface* of Java 9+ applications to assess DARCY$^+$'s ability to enhance security. More specifically, the attack surface of a system is defined as the collection of points at which the system's resources are externally visible or accessible to users or external agents. Manadhata et al. introduced an attack-surface metric to systematically measure the security of a system [155, 156, 157]. Every externally accessible system resource contributes to a system's attack surface, as it can potentially be part of an attack. This contribution reflects the likelihood of each resource being used in security attacks. Intuitively, the more actions available to a user or the more resources accessible through these actions, the more exposed an application is to security attacks [155, 156, 157].

The main resource of a Java 9+ application is Java modules and their packages. As a result, the application's attack surface can be defined as the number of packages accessible from outside its modules. Recall that to measure the attack surface of Java 9+ applications, we count the number of packages exposed by *exports (to)* and *open(s to)* directives since these directives make the internals of packages accessible to other modules (refer to Chapter 3.5.3.)

To evaluate DARCY$^+$'s ability to enhance the applications' security at run-time, we measured the attack surface reduction ratio followed by repairing the inconsistencies of the dynamically loaded modules at run-time. Therefore, for each Java application in our dataset, we selected a module and ran DARCY$^+$ on the rest of the application's modules. Then, we ran the repaired application without that specific module. While running, we dynamically loaded the chosen module and measured the attack surface ratio reduction in the whole application as the result of DARCY$^+$'s repairing the loaded module at run-time. This ratio indicates to what extent DARCY$^+$ is able to further reduce the attack surface of a transformed application by resolving inconsistent dependencies of a dynamically loaded module. We repeated this experiment for all modules in each application and reported the average attack surface reduction ratio per module. As described in Table 5.2, DARCY$^+$ was able to reduce the attack surface of the loaded modules in 36 out of 38 applications by an average of about 17% at run-time. DARCY$^+$ entirely eliminated the attack surface of the loaded module in 28 applications.

Table 5.2 also includes the attack surface reduction ratios followed by DARCY$^+$'s repairs at build-time, which was previously discussed in Chapter 3.5.3. As shown in Table 5.2, 36 out of 38 applications had an average attack-surface reduction of about 56% at their build-time. Note that the average attack surface reduction ratios are usually less than those of build-time as the run-time measurement includes the effects of repairing only one module on the whole application attack surface, whereas at build-time we measured the effects of repairing all modules.

Figure 5.2 shows a cumulative distribution function (CDF) of the ratio of attack surface reduction by DARCY$^+$ among the applications in our dataset at run-time, along with the corresponding results of build-time. It

Table 5.2: Result for Attack-Surface Reduction at Build and Run-time

Application Name	App's Attack Surface Reduction (%)	
	Build-time	Run-time (Avg. Per Module)
Auto-sort	50.00%	7.50%
Ballerina-lang	45.64%	3.09%
Blynk-Server	44.83%	4.02%
BunnyHop	87.50%	7.19%
Codersonbeer-app	75.00%	41.67%
Constantin	100.00%	100.00%
Eclipse Jetty	41.88%	5.14%
Java-9-lab	33.33%	3.13%
Java-9-modularity	-	-
Java-Bookstore	-	-
Java-SPI	50.00%	33.33%
Java9-demo	50.00%	11.43%
Java9-junit	25.00%	16.67%
Java9-labs	80.00%	31.25%
java9-modules	50.00%	3.37%
Java9-TLB-modules	20.00%	2.92%
JavaUtils	85.71%	19.28%
Jigsaw-resources	50.00%	5.00%
Jigsaw-tst	33.33%	6.00%
Jwtgen	100.00%	3.57%
Logback	52.00%	4.82%
Meetup	75.00%	9.38%
Music-UI	60.00%	1.90%
Number-to-text	25.00%	22.22%
Practical-Security	25.00%	18.75%
Quasar	77.27%	24.31%
QuestDB	63.46%	9.75%
Rahmnathan-utils	100.00%	42.86%
RecolInline-Server	47.06%	1.01%
Rhizomatic-IO	15.38%	0.61%
Sense-nine	83.33%	50.00%
Sirius	21.43%	0.70%
Spring-mvn-java9	100.00%	7.14%
Springuni-java9	100.00%	33.33%
Tascalate-Javaflow	45.45%	1.98%
The-Message	10.00%	15.79%
TRPZ	40.00%	50.00%
Vstreamer	66.67%	10.71%
Avg. Attack Surface Reduction	**56.37%**	**16.94%**

indicates that in about 90% of the applications, DARCY$^+$ was able to reduce the applications' attack surface up to 40% on average per loaded module at run-time. Recall that the attack surface reduction at build-time is up to about 50% in 60% of the applications. The relatively large reduction of the attack surface in applications

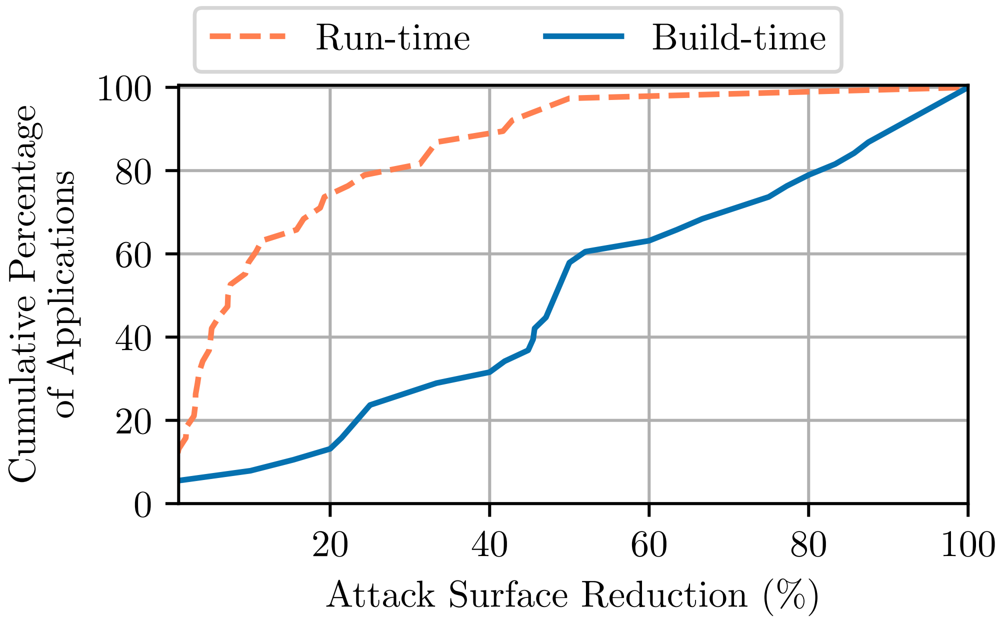


Figure 5.2: Cumulative distribution function (CDF) of the attack surface reduction at build- and run-time

achieved by DARCY$^+$ indicates that it can significantly curtail security risks in Java 9+ applications.

5.5.3 RQ3: Encapsulation

Similar to Chapter 3.5.4, to evaluate DARCY$^+$'s ability to enhance the encapsulation of Java 9+ applications, we leveraged two metrics, i.e., the Ratio of Coupling (RoC) and the Normalized Cumulative Component Dependency (NCD), selected from an extensive investigation by Bouwers et al. [72] about the quantification of encapsulation for implemented software architectures.

The Ratio of Coupling (RoC) [74] measures coupling among an application's modules. For Java 9+ modules, RoC is the ratio of the *number of existing dependencies among modules* to the *number of all possible dependencies among modules*. The Normalized Cumulative Component Dependency (NCD) [143] is the sum of all modules' outgoing dependencies, divided by the total number of modules. For Java 9+ modules, outgoing dependencies are *requires* and *uses* dependencies of each module. Ideally, the value of both RoC and NCD would be low. More specifically, a low RoC value shows that only a small part of all possible dependencies among modules is actually utilized—making it less likely that faults, failures, or errors introduced by changes or additions to modules will propagate across modules. Additionally, a low NCD value indicates lower coupling and better encapsulation.

Table 5.3 presents the amount of RoC and NCD change in 38 Java 9+ applications with inconsistent dependencies at both build- and run-time. These results for build-time were previously showed in Chapter 3.5.4.

To measure the encapsulation improvement at run-time, we ran the same experiment as RQ2. More specifically, we selected a module in each application and ran the transformed application without the chosen module and later at run-time we dynamically loaded the selected module. We then measured the amount of RoC and NCD change in the whole application followed by only repairing the loaded module. We repeated the same experiment for all modules and reported the average RoC and NCD change per module repair for each application. As shown in Table 5.3, DARCY$^+$ was able to reduce the amount of RoC by an average of about 7% and up to about 25% per module in the whole application. The repairs by DARCY$^+$ also reduced the amount of NCD in 24 applications at run-time by an average of about 4% and up to 21%.

At build-time, across all 38 applications, the amount of RoC was reduced by an average of about 28%, and up to about 81%. The amount of NCD was also reduced in 24 applications by an average of about 21%, and up to 79% at build-time. Note that the RoC and NCD change at run-time ratios are less than those of build-time since at run-time, we only measure the effect of repairing a single module on the whole application. These results indicate that DARCY$^+$ can successfully enhance the encapsulation of Java 9+ applications by a significant amount at both build- and run-time.

Figure 5.3 shows the cumulative distribution function (CDF) of the RoC reduction followed by the DARCY$^+$'s repairs at both build- and run-time. It indicates that, in about 70% of applications, DARCY$^+$ was able to reduce the RoC up to 40% at build-time. It also shows that in about 80% of applications, DARCY$^+$ was able to reduce the RoC of the entire application up to about 10% by resolving the inconsistencies of a single module loaded at run-time.

Figure 5.4 demonstrates the cumulative distribution function (CDF) of DARCY$^+$'s NCD reduction rate at both build- and run-time. It shows that resolving the inconsistencies in 80% of the applications could reduce the

Table 5.3: Results for Encapsulation Improvement at Build- and Run-time

Application Name	App's RoC Change (%)		App's NCD Change (%)	
	Build-time	Run-time	Build-time	Run-time
Auto-sort	7.69%	2.00%	-	-
Ballerina-lang	21.43%	1.41%	7.46%	0.53%
Blynk-Server	21.31%	2.74%	-	-
BunnyHop	60.71%	5.21%	25.00%	2.78%
Codersonbeer-app	30.77%	12.05%	12.50%	3.13%
Constantin	55.56%	23.33%	20.00%	5.00%
Eclipse Jetty	35.59%	4.46%	34.18%	4.26%
Java-9-lab	6.67%	1.25%	-	-
Java-9-modularity	9.09%	2.56%	12.50%	3.02%
Java-Bookstore	17.65%	5.07%	16.67%	4.42%
Java-SPI	15.38%	5.58%	5.56%	1.39%
Java9-demo	10.00%	4.40%	-	-
Java9-junit	7.69%	5.13%	-	-
Java9-labs	40.00%	11.90%	-	-
java9-modules	20.00%	2.45%	-	-
Java9-TLB-modules	8.33%	1.64%	-	-
JavaUtils	80.56%	20.86%	79.31%	20.37%
Jigsaw-resources	20.00%	3.13%	-	-
Jigsaw-tst	9.09%	2.31%	-	-
Jwtgen	15.38%	3.01%	8.33%	1.50%
Logback	46.15%	5.48%	21.43%	2.75%
Meetup	42.86%	6.82%	-	-
Music-UI	26.67%	2.68%	10.00%	1.27%
Number-to-text	9.09%	7.60%	-	-
Practical-Security	10.00%	5.00%	7.69%	1.92%
Quasar	52.38%	16.16%	25.00%	6.73%
QuestDB	52.31%	8.55%	8.33%	2.14%
Rahmnathan-utils	50.00%	16.98%	12.50%	3.17%
RecolInline-Server	47.62%	1.07%	48.00%	0.77%
Rhizomatic-IO	5.17%	0.27%	2.44%	0.13%
Sense-nine	29.03%	6.93%	16.00%	2.96%
Sirius	24.32%	1.03%	26.09%	0.90%
Spring-mvn-java9	44.44%	5.29%	16.67%	1.64%
Springuni-java9	50.00%	25.00%	40.00%	16.67%
Tascalate-Javaflow	30.36%	1.66%	26.67%	1.41%
The-Message	6.25%	12.00%	-	-
TRPZ	10.53%	7.54%	-	-
Vstreamer	16.00%	3.66%	20.00%	2.26%

RoC	Total # of Affected Systems	38
	Build-time Reduction Avg.	**27.53%**
	Run-time Reduction Avg.	**6.69%**
NCD	Total # of Affected Systems	24
	Build-time Reduction Avg.	**20.93%**
	Run-time Reduction Avg.	**3.80%**

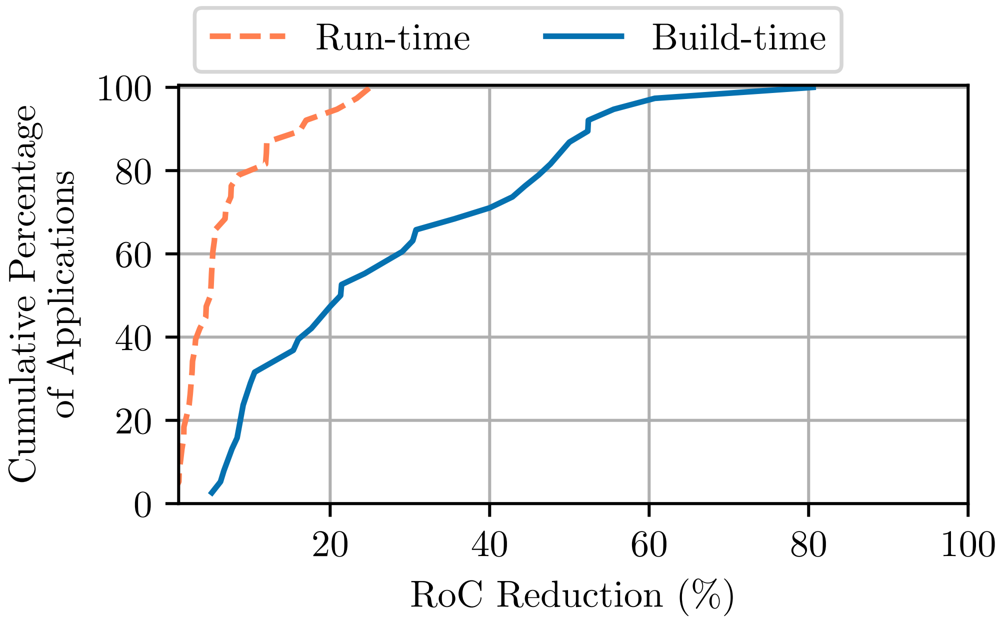


Figure 5.3: revisedCumulative distribution function (CDF) of the Ratio of Coupling (RoC) reduction at build- and run-time

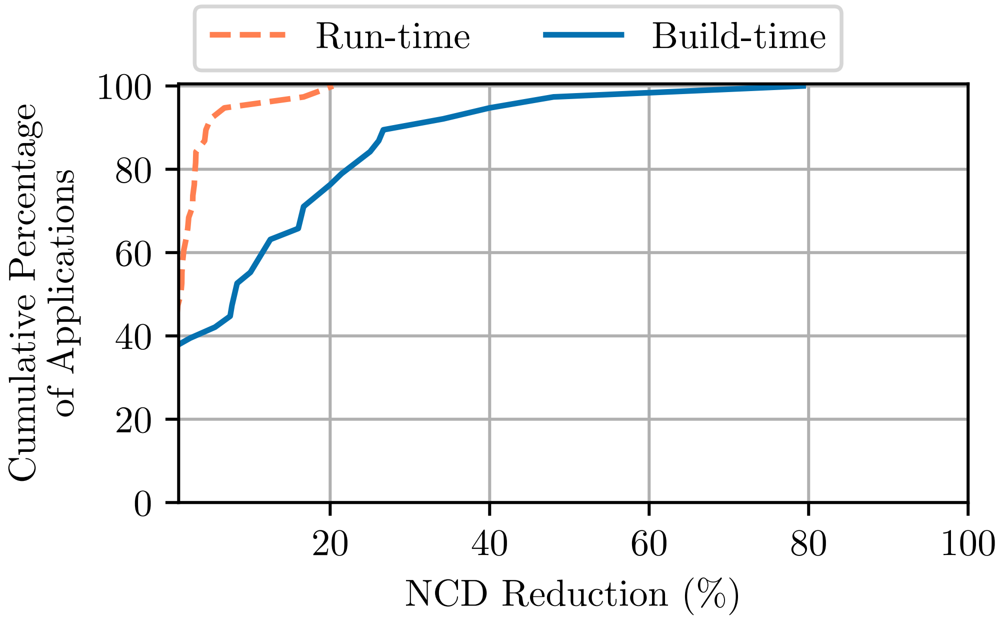


Figure 5.4: Cumulative distribution function (CDF) of the Normalized Cumulative Component Dependency (NCD) reduction at build- and run-time

NCD up to 22% at build-time. Furthermore, it shows that resolving the inconsistencies of a module loaded at run-time can reduce the amount of NCD in 90% of the applications up to about 10%.

5.5.4 RQ4: Software Bloat

To answer this research question, we measured the memory usage of each application before and after DARCY$^+$'s repair phase at both build- and run-time. Recall that in Java 9+, with the JDK being modularized, we can create a lightweight custom Java Runtime Environment (JRE), reducing software bloat (refer to Chapter 3.2.) More specifically, the size of a custom JRE may be reduced after a repair if the application has inconsistent dependencies of type *requires JDK* (Equation 3.2 of Section 3.3).

Table 5.4: Results for Software-Bloat Reduction at Build- and Run-time

Application Name	Runtime Memory Reduction (%)	
	Build-time	Run-time
Ballerina-lang	8.29%	0.09%
BunnyHop	54.72%	20.55%
Eclipse Jetty	16.83%	0.59%
Java-SPI	6.04%	1.51%
JavaUtils	21.87%	19.82%
Practical-Security	0.76%	0.19%
Quasar	4.55%	1.14%
Rahmnathan-utils	0.12%	0.04%
RecolInline-Server	50.70%	7.27%
Sense-nine	0.63%	0.22%
Avg. Memory Reduction	16.45%	5.14%

Table 5.4 shows the reduction of software bloat in terms of the reduction in the size of the JRE image of affected applications after removing inconsistent *requires JDK* dependencies at both build- and run-time repair. According to the results, the reduction is about 16% in 10 applications and up to 55% once the repair is executed at build-time. The table also shows that DARCY$^+$ reduces the application's memory consumption by an average of 5% in those 10 applications after repairing a loaded module at run-time.

Figure 5.5 demonstrates the cumulative distribution function (CDF) of DARCY$^+$'s software bloat reduction rate at both build- and run-time. At build-time, the figure shows that in 80% of the applications DARCY$^+$ could reduce the size of the JRE image up to about 22% by resolving the module inconsistencies. Furthermore, the figure shows that resolving the inconsistencies of a module loaded at run-time can reduce the size of JRE image in 80% of the applications up to about 7%. Such results are particularly substantial for deployment

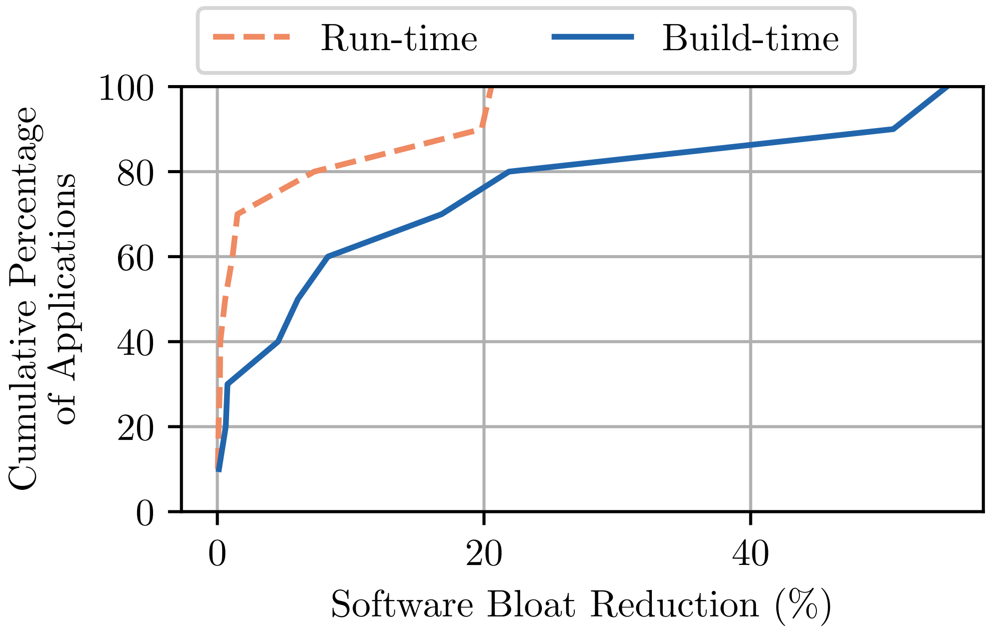


Figure 5.5: Cumulative distribution function (CDF) of software bloat reduction at build- and run-time

and scalability goals in microservices or IoT devices that contain very little memory.

5.5.5 RQ5: Performance

As described in Section 5.4, DARCY$^+$ builds on three tools, Classycle[106], Soot [200], and ANTLR [10]. As a result, similar to Chapter 3.5.6, we assess DARCY$^+$'s performance in terms of these underlying tools' as well as DARCY$^+$'s execution time.

Table 5.5: Results for Execution Time

Component	Avg. Execution Time (ms)	
	Build-time	Run-time
Class Dependency Analyzer (Classycle)	7428	-
Java Reflection Analysis	328	131
ServiceLoader Usage Analysis	315	101
Java Inconsistency Analysis	250	27
Repair	453	163
Total	8774	422

Table 5.5 describes the average execution times for DARCY$^+$ at both build- and run-time. Results for Classycle are shown separately from results for other components since Classycle dominates the execution time. Note

that Classycle is only executed once before applications' run-time, and DARCY$^+$ will store the collected package dependencies for further analysis. Therefore, it only appears in build-time execution time in Table 5.5. On average, as mentioned in Section 3.5.6, DARCY$^+$ takes under 9 seconds for any system to execute at build-time, which is highly time-efficient for both detection and repair. On the other hand, at run-time, DARCY$^+$ takes only about 0.4 seconds on average to detect and repair the inconsistencies of a dynamically loaded module, which causes only a minimal overhead to the system's execution.

5.6 Threats to Validity

There are several threats to internal and external validity that applies at both build- and run-time. Chapter 3.6 discusses these threats to validity in detail, which we briefly explain in this section.

Regarding threats to internal validity, DARCY$^+$ inherits the risk of false positives or negatives from the static analysis techniques it leverage, which might compromise the accuracy of DARCY$^+$'s results. Accordingly, DARCY$^+$ might miss some inconsistencies in the detection phase due to the false positives or negatives of the static analysis tools used in its implementation, e.g., Classycle and Soot. However, both Classycle and Soot are widely used in developing many state-of-the-art tools over the years [184, 220, 89, 147, 114, 150, 88, 130, 144]. Soot has also been actively maintained [19]. Furthermore, we manually examine all detected inconsistencies to avoid any compromise in DARCY$^+$'s accuracy due to unexpected issues with the underlying static analysis techniques.

A possible threat to DARCY$^+$'s validity is the selection of Java applications in our dataset. To minimize this risk, we selected open-source Java 9+ applications from many developers and thousands of repositories in Github, one of the largest and most widely used open-source repositories online. Another threat to external validity is the comprehensiveness of our defect model of module inconsistencies. To mitigate this risk, we studied all types of module directives defined in Java and their possible inconsistencies.

Furthermore, DARCY$^+$ may mistakenly detect some module inconsistencies without knowing developers' or architects' intentions. For instance, in the case of Java libraries, developers may intentionally export or open some packages for further usage, and without being aware of this intention DARCY$^+$ will detect them as inconsistencies. In these cases, DARCY$^+$ takes the developers' input and overrides its decision before repair to resolve this threat.

5.7 Related Work

Most of the related works of DARCY$^+$ have been discussed in Chapter 3.7. In this section, we only focus on related studies to DARCY$^+$'s run-time detection and repair of architectural inconsistencies.

Regarding run-time architectural conformance, in the most closely related work, Yan et al. [219] introduce DiscoTect, a technique that observes running software systems and constructs an architectural view of them. Using this dynamic architectural discovery, developers and architects can map the architectural styles to implementation styles which helps them with building systems consistent with their architectural design. However, their approach does not provide an automated technique to enforce the conformance between prescriptive and descriptive architecture as DARCY$^+$ does.

Overall, DARCY$^+$ is the first approach that studies architectural-implementation conformance, at both build- and run-time, in a mainstream programming language using its own built-in architectural constructs. Additionally, DARCY$^+$ repairs the detected inconsistencies instead of only detecting them. DARCY$^+$ is the first architecture-implementation conformance approach that attempts to reduce the attack surface and software bloat in software applications.

5.8 Discussion

To summarize, we formally defined 8 types of architectural inconsistencies in Java 9+ applications and presented DARCY$^+$, a fully-automated framework that effectively detects and robustly repairs the existing architectural inconsistencies both statically at build-time and dynamically at run-time. For this purpose, DARCY$^+$ leverages static custom analysis, state-of-the-art static analysis tools, and a custom parser generator. According to our experiment, DARCY$^+$'s repair significantly reduces the attack surface, enhances encapsulation, and reduces memory usage for Java 9+ applications. As a future direction of this research, we can provide run-time architectural visualization showing the architectural inconsistencies as they occur. Furthermore, DARCY$^+$ can be expanded as an interactive framework that takes the developer's or architects' feedback and modifies the repair decisions in real-time. DARCY$^+$ can also be implemented as a plug-in for Integrated Development Environments (IDE), which helps developers avoid architectural inconsistencies when developing Java applications.

Chapter 6

Automated Support for Architecture-based Adaptability in Modern Java Applications

Software architecture has been shown to provide an appropriate level of granularity for the representation of a managed software system and reasoning about the impact of adaptation choices on its properties. Software architecture-based adaptability is the ability to adapt a software system in terms of its architectural elements, such as its components and their interfaces. Despite its promise, architecture-based adaptation has remained largely elusive, mainly because it involves heavy engineering effort to make non-trivial changes to the manner in which a software system is implemented. To that end, the incorporation of extensive support for architecture-based development in *Java Platform Module System (JPMS)* inspires realizing architecture-based adaptation capabilities in Java applications. In this chapter, we present ACADIA—a framework that automatically enables architecture-based adaptation of practically any Java 9+ application for *free*, i.e., without requiring any changes to the implementation of the application itself. ACADIA builds on JPMS, which has brought extensive support for architecture-based development to Java 9 and subsequent versions. ACADIA extends JPMS with the ability to provide and maintain a representation of an application's architecture and make changes to it at runtime. The results of our experimental evaluation, conducted on three large open-source Java applications, indicate that ACADIA is able to efficiently apply dynamic changes to the architecture of these applications without requiring any changes to their implementation.

6.1 Introduction

Development of (self-)adaptive software systems is known to be significantly more complex than that of non-adaptive systems [82]. Conventional wisdom in software engineering suggests that a software system's architecture provides an appropriate level of abstraction to mitigate the complexity of adaptive systems [171, 84, 190, 170, 192]. A software system's architecture represents the elements comprising the system (e.g., components, connectors, interfaces) and their relationships to one another [193]. *Architecture-based adaptability* is thus the ability to maintain an accurate representation of a software system's architecture at runtime and effect adaption choices in terms of its architectural elements (e.g., (un)load components).

While the research community has produced many frameworks for architecture-based adaptation (e.g., Rainbow [115] c2.fw [94], ArchJava [58], and Prism-MW [154]), architecture-based adaptation has remained largely elusive in industry. The existing architecture-based adaptation frameworks require the managed software to be designed and developed specifically for adaptation on top of those frameworks. Case in point, in ArchJava [58], developers would need to implement their Java applications using syntax that is specific to ArchJava. For example, a component in ArchJava is defined using a special "component" keyword before class declaration (`public component class nameOfClassHere`), while a component's interfaces are defined using a special "port" keyword before method declaration (`public port nameOfMethodHere`). Despite their elegance, the above-mentioned frameworks are not used for architecture-based adaptation of real-world software, because developers do not build their systems on top of these framework.

Indeed, the majority of real-world software is not developed architecturally. Widely-used programming languages and frameworks do not provide explicit support for architectural concepts. In other words, there is a gap between architectural constructs (e.g., components, connectors, interfaces) used to conceptualize the architecture of a system and the low-level programming constructs (e.g., classes, methods, variables) used for their implementation. A system that is not developed architecturally is practically impossible to adapt without a significant engineering effort. At a minimum, developers have to recover what parts of the code map to architecturally relevant elements, develop the ability to dynamically (un)load and (un)bind those elements, and further implement the necessary facilities to keep that representation of the software in sync with the running system.

These difficulties have also impacted software engineering researchers that have struggled with the lack of real-world adaptive software for proper evaluation of their techniques. Researchers have responded to this problem by developing a repository of exemplar adaptive applications to help with validation of their work [43]. The majority of these exemplars, however, are essentially still toy applications. Indeed, to this date, the majority of publications in this domain, including those of the authors, evaluate the reported techniques on either small, handcrafted applications or simulated environments. There are no effective means of comparing the techniques on the same set of real-world applications, similar to the manner researchers evaluate their work in other areas of software engineering, for instance software testing. The lack of access to real-world software readily available for experimentation has truly hindered research in this area.

Inspired by relatively recent developments in Java, which for the first time has incorporated extensive support for architecture-based development, we believe the software engineering community is finally at the cusp of realizing architecture-based adaptation for *free*, i.e., without requiring the developers to implement their systems on top of a separate framework for architecture-based adaptation. In its 9th iteration, Java has introduced the Java Platform Module System (JPMS) [8], which allows developers to explicitly specify the system's software components (i.e., modules in JPMS), their interfaces, and the specific nature of their dependencies in a file called `module-info.java`.[1] JPMS creates opportunities for building adaptive software systems in ways that were not possible before. Specifically, the architectural elements of any Java application built in Java 9 and subsequent versions are readily attainable from its `module-info.java` file.[2]

Despite its support for architecture-based development, JPMS does not provide a mechanism for maintaining a runtime model of the system's architecture and modifying it at runtime. Accordingly, JPMS by itself is lacking support for architecture-based adaptation of Java applications. In this chapter, we present ACADIA, a framework that allows for any Java 9+ software system to be adapted architecturally. ACADIA leverages JPMS features and static analysis to (1) determine the architecture of Java applications and (2) automatically transform the applications to a form that can be adapted at runtime. Using ACADIA, developers can determine or update their Java applications' architecture at runtime without requiring any additional implementation or any modification to their applications. Moreover, ACADIA is extremely efficient, enabling developers to

[1]Unfortunately the notion of software connector is still not explicit in JPMS.
[2]We may refer to Java 9 and subsequent versions as Java 9+ in this chapter.

execute various adaptation strategies with very low overhead. An implementation of the proposed technique is publicly available [51]. ACADIA offers profound and practical contributions to the software engineering research community by making it possible to evaluate the techniques developed in this community on hundreds of open-source Java 9+ applications with minimal effort. Researchers can further empirically compare their techniques and replicate prior studies on the same set of real-world, adaptive software, thereby advancing the research in this area.

The remainder of this chapter is organized as follows. Section 6.2 provides background on the Java module system, including JPMS programming constructs and its design goals, and illustrates an example modularized Java application. Section 6.3 describes ACADIA and its implementation in detail. Section 6.4 demonstrates an application of ACADIA in terms of an example adaptation strategy built upon it. Section 6.5 presents the experimental evaluation of ACADIA's adaptation capabilities. The chapter concludes with an outline of related research.

6.2 Architecture-based Development in Java

In this section, we first introduce the concepts behind the Java Platform Module System (JPMS) [8], followed by the description of an illustrative example used throughout the chapter for explanation of our work.

6.2.1 Java Platform Module System

JPMS allows software developers to specify the architecture of Java applications in terms of software components, component interfaces, and dependencies among the application's components. A Java *module* in JPMS represents a software component, and it is a uniquely named reusable group of related packages and resources.

The main goals of modular development in JPMS include reliable configuration, strong encapsulation, scalability, platform integrity, and improved performance [96]. JPMS enables developers to explicitly specify module interfaces and dependencies. In other words, developers can identify the public and private members of a module that are accessible or inaccessible to other modules. As JPMS provides a more refined accessibility control, it significantly increases Java applications' encapsulation. Furthermore, Java has modularized the JDK itself using JPMS modules. As a result, developers can create lightweight custom Java Runtime Environment (JRE) images consisting of only modules they need for their application, which improves the scalability and performance of Java applications [6].

Each Java module includes a *module descriptor file* specifying the module's interfaces and dependencies to other modules. Specifically, the descriptor file, called `module-info.java`, includes the module name, other modules it depends on, the packages it explicitly makes available to other modules, the services it offers, the services it consumes, and to what other modules it allows reflection [96]. A module can utilize combinations of the following five module directives to describe the details of its interfaces and dependencies: (1) the *requires* directive, which specifies other modules that a module needs access to, (2) the *exports* directive, which makes public members of a package accessible by other modules, (3) the *opens* directive, which opens public and private members of a package to reflective access by other modules, (4) the *provides* directive, which specifies the services a module provides, and (5) the *uses* directive, which specifies the services a module consumes [135].

6.2.2 Illustrative Example

Figure 6.1 illustrates an example Java application modularized using JPMS. It consists of three modules *User*, *Util*, and *Network*. Each module's descriptor file, i.e., `module-info.java`, is shown in the figure. Accordingly, module *User* contains two packages, *foo* and *bar*, and reads modules *Util* and *Network* in its implementation, specified by `requires` directives. Module *Util* contains two packages, *core* and *api*. It reads module *Network* in its implementation, specified by `requires` directive, and uses the service `NetService`, specified by `uses` directive. Module *Util* also exposes its *core* and *api* packages to be accessible by other modules, using `exports` directives, and opens its *core* package specifically for reflective access to the module *User*, using `opens` directive. Module *Network* contains a package, called *net*, and provides an implementation of the service *NetService* in one of its classes, called *NetClass*, which is specified by *provides* directive.

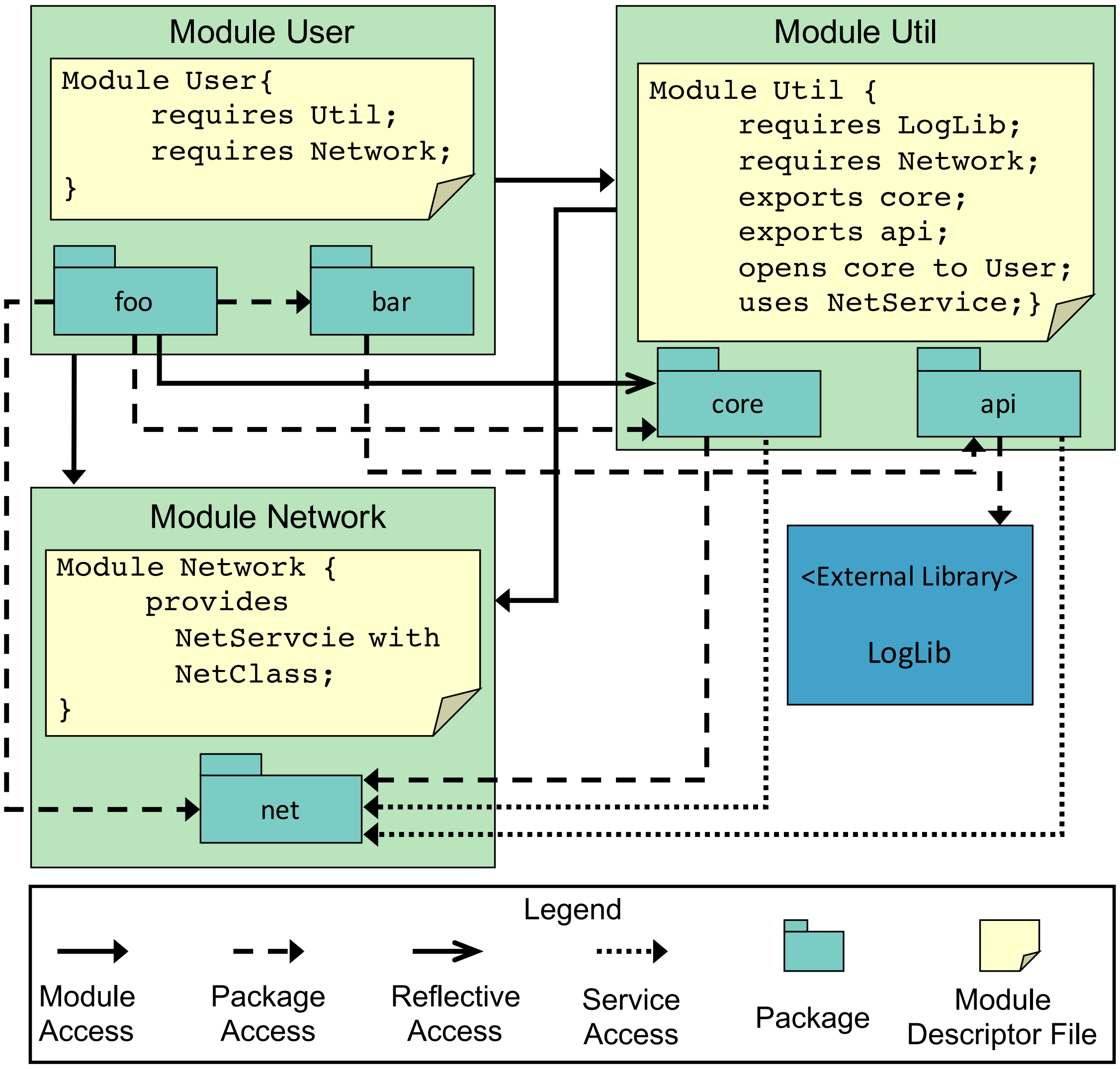


Figure 6.1: An example Java app implemented using JPMS

These directives outline an overview of the dependencies between the system's modules and their packages. Figure 6.1 shows these dependencies with arrows. More specifically, package *foo* in module *User* accesses internals of packages *bar*, *core*, and *net* in its implementation. It also reflectively accesses private members of package *core*. Package *bar* accesses the internals of package *api*. Package *core* accesses internals of package *net* and uses the service package *net* provides. Package *api* uses the external library *LogLib* and uses the service provided by package *net*.

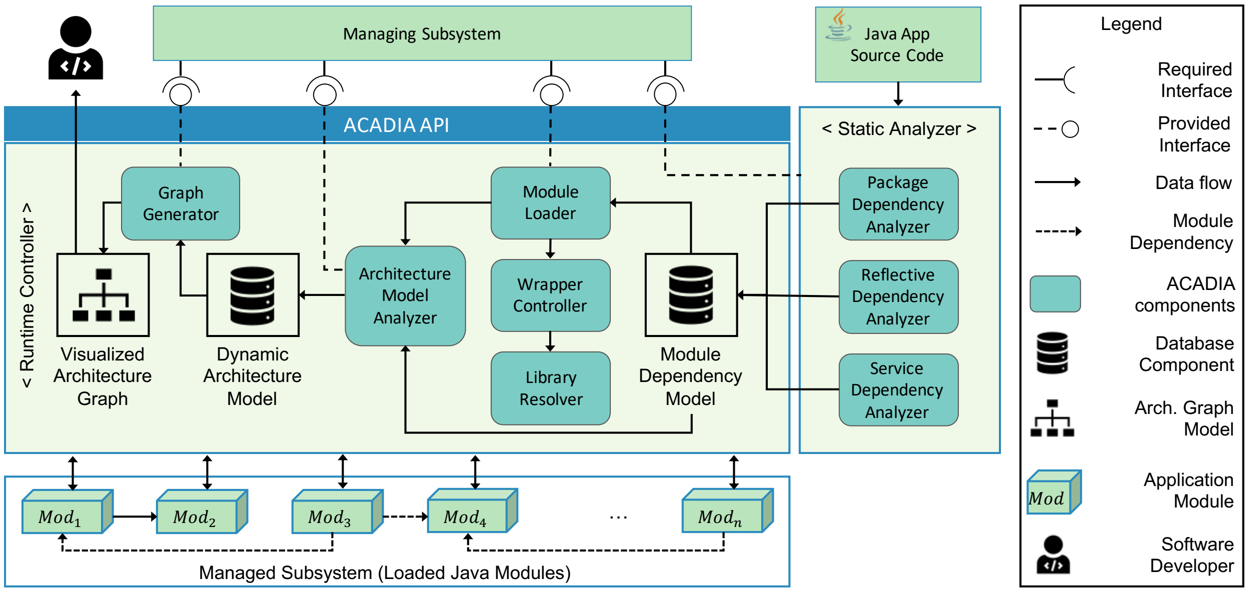


Figure 6.2: A high-level overview of ACADIA framework

6.3 ACADIA: Framework Overview

Figure 6.2 depicts a high-level overview of ACADIA consisting of three main compartments: ACADIA *API*, *Static Analyzer*, and *Runtime Controller*. In this section, we describe how ACADIA enables architecture-based adaptation capabilities in Java 9+ applications by explaining each compartment's implementation in detail.

6.3.1 ACADIA's API

ACADIA takes a Java 9+ application's source code as input and provides a specific API including a set of commands with which any adaptation strategy or software developer can interact. The adaptation strategy is denoted by *Managing Subsystem* in Figure 6.2. More specifically, ACADIA can execute various architecture-based adaptation operations on the running Java system, denoted by *Managed Subsystem* in Figure 6.2. These operations include applying dynamic changes to the running system, obtaining an accurate architectural model of the system at runtime, or visualizing it at any time during the execution of software.

Table 6.1 shows ACADIA's API, represented as a predefined set of commands. These commands can be grouped into five categories: (1) commands for loading and unloading of modules, i.e., `load`, `load-all`, and `unload`, (2) a command for statically analyzing a new module to obtain its architecturally-relevant properties, i.e., `static-analyzer`, (3) commands for modification of dependencies, i.e., `add-requires`, `add-exports`, and `add-opens`, (4) commands for retrieving a system's dynamic architectural model, i.e., `get-arch-model`, `get-dependent`, and `list-all-loaded-modules`, and (5) a command for visualization of the system's dynamic architecture for human perception, i.e., `visualize`. Each category of API commands are implemented by specific components in the *Runtime Controller* compartment, as elaborated further in Section 6.3.3.

Table 6.1: Commands describing ACADIA's API

Command	**Description**
`load` <*moduleName*> <*path*>	Loads the module <*moduleName*> from the path <*path*>.
`load-all` <*path*>	Loads all modules of the Java application from the path <*path*>.
`unload` <*moduleName*> <*path*>	Unloads the module <*moduleName*> from the path <*path*>.
`static-analyzer` <*moduleName*> <*path*>	Runs the *Static Analyzer* on a new module, <*moduleName*> from path <*path*>, to retrieve and store its package dependencies.
`add-requires` <*source*> <*target*>	Makes module <*target*> accessible by module <*source*> through a `requires` directive.
`add-exports` <*source*> <*pckg*> <*target*>	Makes the package <*pckg*> from the module <*source*> accessible to the module <*target*> through an `exports` directive.
`add-opens` <*source*> <*pckg*> <*target*>	Makes the package <*pckg*> from the module <*source*> accessible via reflection to the module <*target*> through an `opens` directive.
`get-arch-model`	Generates the real-time architecture model of the Java app in terms of a graph representation, characterized by different types of dependencies.
`get-dependent` <*Module*> [<*pckg*>]	Retrieves a subset of the architecture model including the packages and modules that are dependent on the module <*Module*> and package <*pckg*>, characterized by different types of dependencies.
`list-all-loaded-modules`	Lists all deployed modules at anytime of the runtime.
`visualize`	Visualizes the architecture graph of the system in a figure for human perception.
`stop`	Stops the ACADIA's framework and quits.

6.3.2 Static Analyzer

Before running the target Java application, ACADIA performs a set of static analyses on the application's source code to acquire the necessary information about the implementation of the Java application. ACADIA relies on an existing static analysis approach, represented as *Package Dependency Analyzer* in Figure 6.2, to identify the dependencies between all packages of a Java application. We leveraged Classycle [106] in the implementation of ACADIA. More specifically, ACADIA retrieves all package dependencies implemented in each Java module by running Classycle before starting the application. Although Classycle collects all information about the system dependencies at both the class level and package level, ACADIA only requires the package-level dependencies, since module directives can only define and manage dependencies between packages.

A Java application might contain packages that reflectively access one another, indicating dependencies of type *opens* directive. This type of reflective dependencies cannot be retrieved by *Package Dependency Analyzer*. For this purpose, we implemented a custom static analysis, using the Soot framework [200], represented as *Reflective Dependency Analyzer* in Figure 6.2. It analyzes the source code of the Java app and extracts all instances of reflective dependencies [116]. Figure 6.3 shows an example of a reflective invocation in Java. A reflective invocation first obtains a private class of interest (line 2), followed by selecting one of the class's methods using its name and parameters (lines 3-6). It finally invokes the selected method (line 7). The *Reflective Dependency Analyzer* component uses a backward static analysis and follows the use-def chain of any `java.lang.reflect.Method` instances to identify all reflective invocations in the source code. It then collects all the selected class and method information in the reflective invocation.

```
 1 try {
 2     Class clazz = Class.forName("className"); ()
 3     Class argz = new Class[2]; ()
 4     argz[0] = Integer.TYPE; ()
 5     argz[1] = Integer.TYPE; ()
 6     Method method = clazz.getMethod("methodName", argz); ()
 7     method.invoke(..., ...); ()
 8 }catch (Throwable e) {
 9     System.err.println(e);
10 }
```

Figure 6.3: An example of a reflective dependency in Java

Furthermore, Java applications can contain implementations of service providers using `ServiceLoader` API, which indicates the dependencies of types *provides* and *uses* directives. Figure 6.4 shows an example of a service dependency in Java [18]. In this example, `CodecSet` is a service class providing two functionalities: `getEncoder` and `getDecoder`. Here, `ServiceConsumer` is a class that aims to consume these functionalities. It uses `ServiceLoader`, a built-in Java class, to load the provider classes of the corresponding service (line 6). It then iterates over possible provider classes and chooses appropriate ones with the `getEncoder` method (lines 7-9). To identify such dependencies, we developed another custom static analysis, using the Soot framework [200], shown as *Service Dependency Analyzer* in Figure 6.2. It analyzes the source code of the Java application, leverages a backward analysis, and follows the use-def chain of any instances of the `ServiceLoader` class [116].

```
 1 interface CodecSet{
 2     public abstract Encoder getEncoder(String encodingName);
 3     public abstract Decoder getDecoder(String encodingName);
 4 }
 5 Class ServiceConsumer{
 6     private static ServiceLoader<CodecSet> codecSetLoader = ServiceLoader.
      load(CodecSet.class); ()
 7     for (CodecSet cp : codecSetLoader) { ()
 8          Encoder enc = cp.getEncoder(encodingName); ()
 9          if (enc != null) {...} ()
10      }
11 }
```

Figure 6.4: An example of service dependency in Java

The collected information from *Package Dependency Analyzer*, *Reflective Dependency Analyzer*, and *Service Dependency Analyzer* components is stored in *Module Dependency Model*, i.e., a database component, which is later used by the *Runtime Controller* compartment. In addition, in case the *Managing Subsystem* introduces a new module at runtime, for which the module dependency information is not available, it can use the `static-analyzer` command (recall Table 6.1) to invoke the *Static Analyzer*, extract that information, and update the *Module Dependency Model*.

6.3.3 Runtime Controller

The *Runtime Controller* compartment is responsible for the implementation of ACADIA's runtime API, listed in Table 6.1. In the remainder of this section, we explain the implementation details of each category of API commands.

The *Module Loader* component is responsible for loading and unloading modules at runtime, i.e., `load`, `load-all`, or `unload` commands. In case of loading a module into a running system, *Module Loader* component creates a module *wrapper*, i.e., a graph of coherent modules representing the module itself and other modules or libraries it depends on. Upon loading a new module, that module and all unloaded modules that it depends on are loaded into a new wrapper. The wrapper is implemented using `java.lang.ModuleLayer` class in JPMS [33], which defines a group of modules in the Java virtual machine (JVM). When starting a Java application, the Java runtime creates an initial layer called *boot layer*. The boot layer contains the module graph of resolved modules provided when running the Java application. Later, new `ModuleLayers` can be defined to deploy other new modules inside them. A Java application can include multiple layers during its execution, and layers themselves can have dependencies amongst each other [153]. The `java.lang.ModuleLayer` class is only an implementation for a group of Java modules. Whenever a `ModuleLayer` is created, JVM assigns a new instance of `ClassLoader` to that layer which enables loading a new module at runtime. Our evaluation (see Section 6.5) will demonstrate that the performance cost or overhead of using our wrapper implemented using `ModuleLayer` is small and practical.

After creating the wrapper, *Module Loader* determines the dependent modules and libraries using the *Module Dependency Model* database. Then, it invokes *Wrapper Controller*, which is responsible for managing all of the created wrappers in the running system, to register the recently created wrapper. More specifically, once a wrapper is created, *Wrapper Controller* notifies the JVM about the classes that may be loaded from the modules inside that wrapper, such that JVM knows to which module each loaded class belongs. Wrappers can also have dependencies amongst each other in cases where a module required inside a wrapper is previously loaded into another wrapper. Lastly, the *Library Resolver* component loads the required external libraries inside the wrapper. At this point, *Module Loader* can load the target module and its dependent modules into the *Managed Subsystem*. To unload a module, *Module Loader* identifies the target module's dependent modules and libraries. i.e., modules that are only used by the target module. Subsequently, the *Wrapper Controller* locates the module's corresponding wrapper and removes it.

To better explain the loading or unloading process of a module, we use the Java application introduced in Section 6.2.2. Consider a scenario in which the *Managed Subsystem* first starts with no Java modules loaded. Therefore, the Java runtime starts with the boot layer consisting of the necessary JDK modules. Suppose the *Managing Subsystem* enters the command to load module *Util*. To that end, *Module Loader* retrieves its dependent modules and libraries, i.e., *Network* and *LogLib* library. It then creates a new wrapper (Wrapper 1) and loads the *Util* and *Network* modules into it. Next, *Wrapper Controller* registers the recently created wrapper and notifies the JVM. Before loading the entire wrapper into the JVM, *Library Resolver* loads the external library *LogLib* from the provided path. Finally, *Module Loader* loads the entire wrapper into the *Managed Subsystem*.

Next, suppose the *Managing Subsystem* enters the command to load module *User*. Consequently, *Module Loader* determines that the *User* module is dependent on modules *Util* and *Network* using the *Module Dependency Model*. As a result, *Module Loader* creates a new wrapper (Wrapper 2) and loads module *User* into it, and *Wrapper Controller* registers it by notifying the JVM. There is no need to load the dependent modules since they are already loaded into Wrapper 1. Instead, *Wrapper Controller* creates a dependency from Wrapper 2 to Wrapper 1. Figure 6.5 shows the *Managed Subsystem* after ACADIA has loaded all of the modules of this example within two wrappers. Later, suppose the *Managing Subsystem* decides to unload the *User* module. As a result, the *Module Loader* and *Wrapper Controller* components respectively unload *User* and delete *Wrapper 2*, as it contains no other modules.

The *Architecture Model Analyzer* component listens to *Module Loader* or the input commands regarding dependency modifications for any changes to the architecture of the system. It uses the *Module Dependency Model* database to create the system's *Dynamic Architecture Model* and keep it updated. The dynamic architecture model of the system should capture all its modules, packages, and all types of their dependencies. It can be modeled using a graph where the system's modules and packages are represented as graph nodes and their dependencies as graph edges. However, there are different types of dependencies between packages and modules, specified by various module directives in JPMS. Therefore, the dynamic architecture cannot be

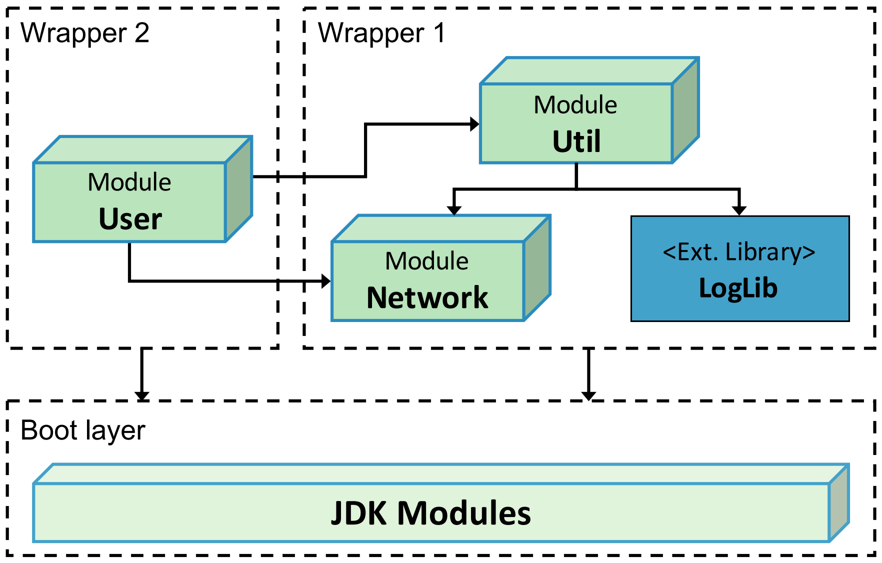


Figure 6.5: The managed subsystem after loading all modules of the example Java app (Section 6.2.2)

modeled with a single graph, as there might be more than one edge between two nodes. For this purpose, we leverage the concept of *multidimensional networks* for modeling graphs with different types of edges [69]. In the context of our architectural model, there are five different types of edges: the *Containment* edge between package nodes and their corresponding module nodes, the *Module Access* specified with `requires` directives between module nodes, the *Package Access* specified with `exports` directives and the package dependencies in the system's implementation, the *Reflective Access* specified with `opens` directives and the reflective dependencies in the system's implementation, and the *Service Access* specified with `provides` and `uses` directives between packages.

DEFINITION 6.1. *The dynamic architecture model of the managed subsystem can be formally specified as a triple*

Dynamic_Architecture_Model $\equiv \langle V,\ D,\ E\rangle$*, where*

- V *is the set of nodes including the loaded Java modules, all their packages, and the external libraries,*
- $D = \{Containment, ModuleAccess, PackageAccess, ReflectiveAccess, ServiceAccess\}$ *is the set of dimensions specifying different types of dependencies between the nodes,*

- *$E = \{(u,v,d) | u,v \in V \ \wedge d \in D\}$ is the set of dependencies between the nodes.*

Figure 6.6 demonstrates the multidimensional network representation of the architectural model of our illustrative example application, explained in Section 6.2.2, after loading all of its modules. As shown in the figure, there might be more than one edge between two nodes in a multidimensional network.

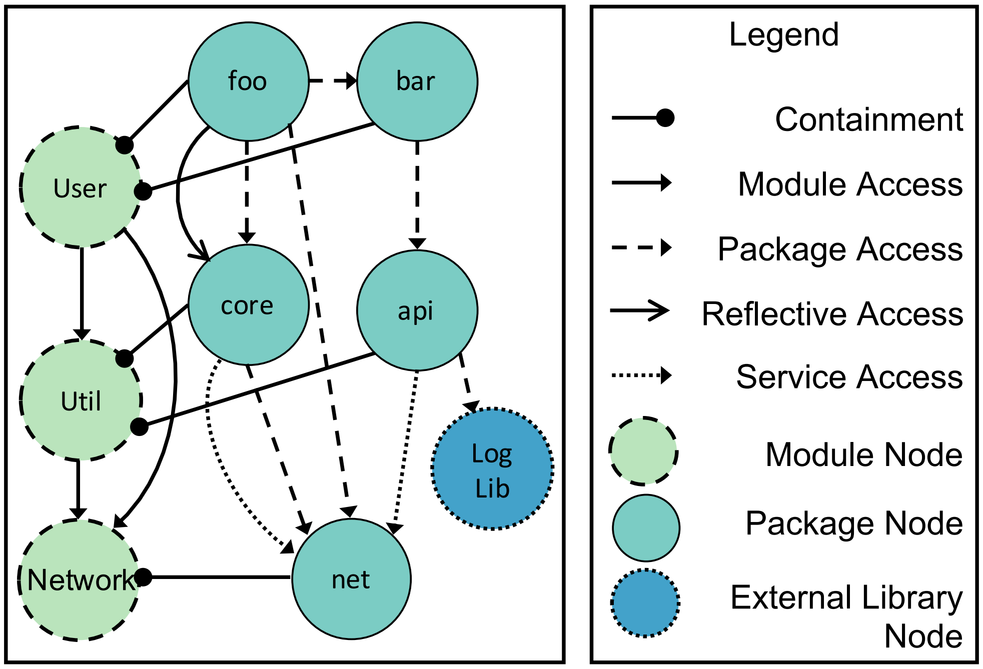


Figure 6.6: The multidimensional network representing the dynamic architecture model of the illustrative example in Section 6.2.2 after loading all modules

The *Architecture Model Analyzer* is responsible for generating and updating the architectural model upon the occurrence of any changes to the architecture of the *Managed Subsystem* by adding or removing nodes or edges, as a result of (un)loading modules and modifications to their dependencies. Moreover, in the case of `get-dependent` command, ACADIA needs to identify the dependent modules and packages to a specific module or package. To that end, *Architecture Model Analyzer* defines this search as a connected component search problem in the context of a multidimensional network [69]. In other words, the dependent nodes on a specific node in a specific set of dimensions can be formally specified, as shown in Equation 6.1.

$$Dependent\_Nodes(v,D) \equiv \{v \in V \mid \exists(u,v,d) \in E \wedge d \in D\} \tag{6.1}$$

The *Graph Generator* component returns the architectural model of the system in response to `get-arch-model` command. To that end, the *Graph Generator* converts the multidimensional network representation of the system's architectural model to a graph description file in DOT language [42], i.e., a widely-used graph description language. Furthermore, in the case of the `visualize` command, it uses visualization libraries, such as Graphviz [112], to visualize the generated graph in DOT language for human perception.

6.4 Application of ACADIA

To evaluate ACADIA, we used it to develop a self-adaptation logic aimed at keeping the memory usage of Java 9+ applications below a user-provided threshold. Memory efficiency is particularly critical for deployment of Java-based applications on resource-constrained devices, e.g., embedded and IoT devices.

While the application is running, the adaptation logic monitors its runtime memory usage. Once it gets larger than a predefined threshold, it attempts to reduce the system's memory footprint by either (1) offloading unnecessary modules, i.e., application modules that are not being used at the moment, or (2) replacing some modules with their alternative memory-efficient versions, if available.[3] Later, if a previously offloaded or replaced module is needed again, the adaptation component reloads it using the framework. The details of this adaptation strategy are presented in Algorithm 3. It uses ACADIA's API commands to execute its adaptation operations.

Algorithm 3 takes the Java application (App) and the user-defined memory threshold ($MemoryThold$) as inputs. While the app is running, it continuously monitors the app's memory usage, and if it goes above the threshold, it applies the proper adaptation strategies aimed at reducing its memory usage. More specifically, whenever the runtime memory size gets larger than the predefined threshold *MemoryThold*, Algorithm 3 first retrieves the list of running methods using the stack trace of the app's running thread, and consequently,

[3]It is often possible to realize a given functionality with different degrees of memory consumption based on the implementation choices involving data structures (e.g., arrays vs. vectors vs. linked lists), storage locations (e.g., random access memory vs. hard drive), and algorithms. Of course, such choices may also affect other quality attributes, e.g., execution speed. For the sake of simplicity, we did not take into account the effect of adaptation choices on other quality attributes.

Algorithm 3: Adaptation Strategy

```
   Input: $App$ // The Java Application to run
   Input: $MemoryThold$ // Memory size threshold
1  while $app.thread.isAlive()$ do
2      if $getRuntimeMemory() > MemoryThold$ then
3          $RunningModules \leftarrow$ getModules($app.thread.getStackTrace()$)
4          foreach $module \in App.loadedModules()$ do
5              if $module \notin RunningModules$ and $ACADIA.getDependent(module) \notin RunningModules$ then
6                  ACADIA.unload($module$)
7              end
8              if $module.isLoaded()$ and $hasReplacement(module)$ then
9                  $NewModule \leftarrow module.getReplacement()$
10                 $DependentModules \leftarrow$ ACADIA.getDependent($module$)
11                 ACADIA.load($NewModule$)
12                 ACADIA.updateDependencies($module$, $DependentModules$, $NewModule$)
13                 ACADIA.unload($module$)
14             end
15             if $getRuntimeMemory() < MemoryThold$ then
16                 Break
17             end
18         end
19     end
20     $RequiredModules \leftarrow$ getModules($app.thread.getStackTrace()$)
21     if $App.loadedModules \neq RequiredModules$ then
22         ACADIA.load($RequiredModules$ - $App.loadedModules()$)
23     end
24 end
```

obtains the list of modules being used and stores it in *RunningModules* (line 3). Then, for each loaded module, the algorithm attempts to either offload or replace it to reduce the runtime memory size (lines 4-18). Trying to offload the module, Algorithm 3 checks whether the module can be offloaded safely. In other words, it checks if the target module is not currently running. Algorithm 3 also uses ACADIA's `get-dependent` command to retrieve the modules dependent on the target module and ensures none of them are among the running modules at the moment (line 5). If true, the algorithm offloads the target module using ACADIA's `unload` command (line 6).

However, in case the target module is being used and cannot be offloaded, Algorithm 3 tries to replace the target module with its memory-efficient version, if any (line 8). For this purpose, the module's replacement is stored in *NewModule* (line 9). Moreover, the algorithm uses ACADIA's `get-dependent` command and obtains the list of modules dependent on the target module (line 10). These dependencies should be updated to reflect the *NewModule*. Therefore, Algorithm 3 first loads the new module using ACADIA's `load` command (line 11) and then modifies the other dependent modules' dependencies to *require*, *export*, or *open* their packages to the new module accordingly (line 12). To that end, the *updateDependencies* method calls ACADIA's `add-requires`, `add-exports`, or `add-opens` commands where applicable. At this point, ACADIA can safely offload the old version of the module as it is no longer used (line 13).

After the consideration of each module for offloading and replacement, the algorithm evaluates whether the runtime memory footprint of the application is less than the *MemoryThold*, and if so, it stops considering the other modules (lines 15-17). If the application requires a previously offloaded module for its execution, the algorithm retrieves the list of required modules using the stack trace of the app's thread again and stores it in *RequiredModules* (line 20). It then checks if any of these required modules are not loaded (line 21). In that case, it uses ACADIA's `load` command to load them (line 22). The algorithm eventually terminates when the application thread stops.

6.5 Evaluation

To evaluate ACADIA, we study the following research questions: **RQ1:** How successful is ACADIA in adapting real-world Java 9+ applications without requiring changes to the manner in which they are implemented?
RQ2: What is ACADIA's performance in terms of its execution time?
RQ3: To what extent does ACADIA impose an overhead on Java 9+ applications' execution time?

To answer these questions, we selected three large non-trivial open-source Java 9+ applications from GitHub [11]. Using these applications, we designed an experiment in which we execute the adaptation strategy explained in Section 6.4 on a MacBook Pro 2013 (2.3 GHz Intel Core i7, 16 GB, MacOS 10.14). In this section, we first introduce the selected Java applications and describe their architecture in detail. We then discuss the mentioned research questions and present the details of the experiments' results.

6.5.1 Subject Applications

To examine the objectives of ACADIA, we used three different open-source Java applications on GitHub [11], namely Quasar [4], Tascalate JavaFlow [9], and Rhizomatic [16]. Each subject application contains multiple Java modules that can be utilized in many example scenarios. In this experiment, we used the example scenarios (use cases) officially introduced by the applications' developers to ensure they truly reflect real-world usages of these applications.

The first application, Quasar, is an open-source Java system that provides high-performance, lightweight threads, Go-like channels, Erlang-like actors, and other asynchronous programming tools for Java and Kotlin. Quasar aims to simplify the development of highly concurrent software applications by adding lightweight threads, channels, or actors to the JVM [14]. Quasar consistes of 4 modules and 33 packages. It also depends on three external libraries. Figure 6.7 shows a high-level overview of Quasar's architecture, including its modules and required libraries.

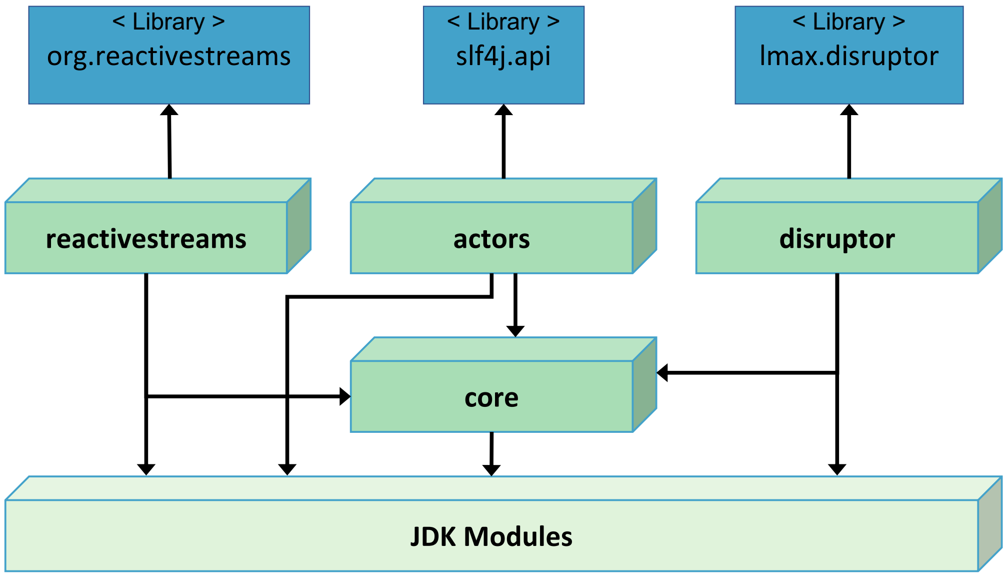


Figure 6.7: Quasar's architecture graph in module-level

The second application, Tascalate JavaFlow, contains libraries and tools for developing Java applications using continuations, i.e., an abstract representation of the control state of a computer program. Tascalate JavaFlow allows developers to capture the execution state of the program. e.g., local variables, execution stack, program counters, etc., at a specific time and later resume the execution from that saved state. [9].

Tascalate JavaFlow is implemented in 10 modules, containing 27 packages in total. Figure 6.8 demonstrates the module-level architecture of Tascalate JavaFlow.

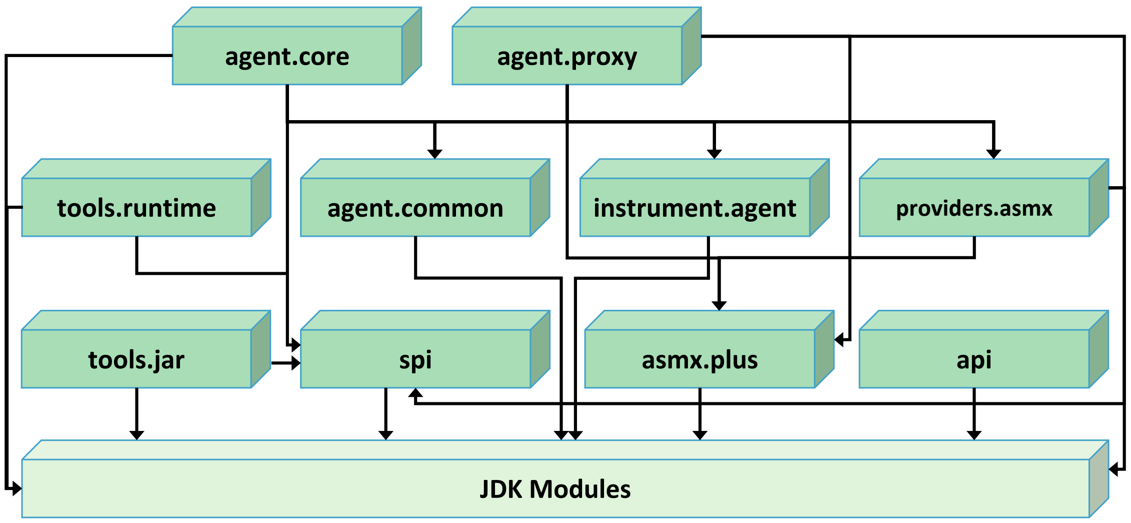


Figure 6.8: Tascalate's architecture graph in module-level

The third application, Rhizomatic, is a runtime environment built on JPMS that provides different programming services. It supports a service programming model based on Google Guice [5] and provides programming models for developing web application servers and RESTful applications in Java [16]. Rhizomatic consists of 7 modules and 157 packages. Figure 6.9 illustrates a high-level overview of Rhizomatic's module architecture.

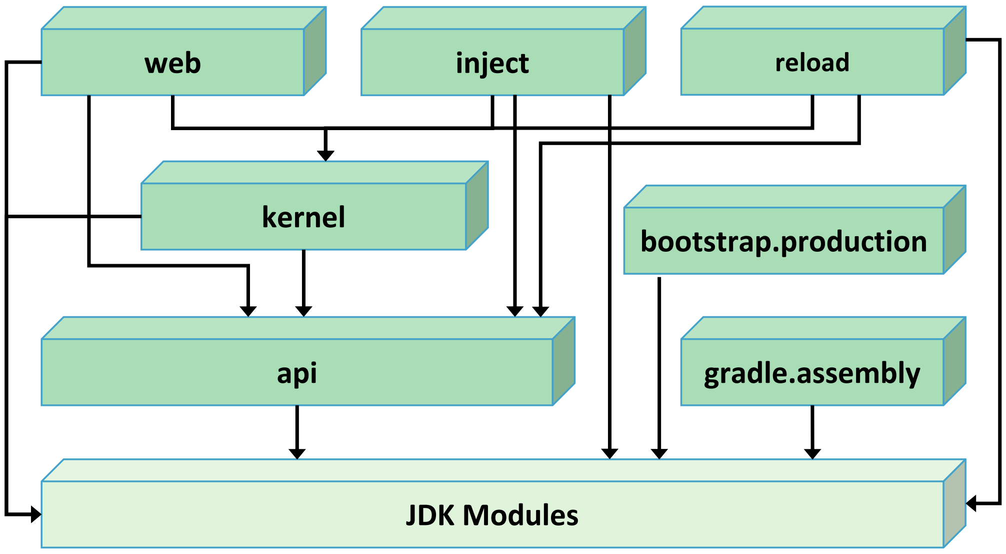


Figure 6.9: Rhizomatic's architecture graph in module-level

6.5.2 RQ1: Adaptation Effectiveness

Table 6.2: Results of ACADIA performing an adaptation strategy on different example scenarios of three subject Java apps

App Name	Example Scenario	AVG. Mem. (MB) Before	After	Δ %	Adapt. Success
Quasar (Max Mem.: 62 MB Mem. Thold: 40 MB)	Fiber	51	24.4	52.16	✓
	FiberAsync	55	28.4	48.36	✓
	IO	59	32.4	45.08	✓
	Channel	52	24.8	52.31	✓
	Actor	56	36.8	34.29	✓
	SrvrBhvr	60	41.8	30.33	✗
	ProxySrvr	52	33.8	35.00	✓
	EventSrc	51	29.6	41.96	✓
	Supervisor	58	35	39.66	✓
	ReactStrms	62	40.6	34.52	✗
	Issues	59	39.8	32.54	✓
	Migration	52	33.4	35.77	✓
	PingPong	54	34.8	35.56	✓
	GenEvent	60	40.8	32.00	✗
	GenServer	54	32.2	40.37	✓
	GalaxySrvr	52	29.6	43.08	✓
Average Runtime Memory Size Reduction:				39.56%	
Adaptation Success Rate:				81%	
Tascalate (Max Mem.: 12 MB Mem. Thold: 6 MB)	cdi-owb	11	3.2	70.91	✓
	cdi-weld3	12	3.2	73.33	✓
	cd-weld4	12	3.2	73.33	✓
	cglib	12	5.4	55.00	✓
	common	12	5.8	51.67	✓
	dyninvoke	12	6.42	46.43	✗
	jee	12	3.2	73.33	✓
	lambdas	11	3.2	70.91	✓
	retro-lmbds	12	3.2	73.33	✓
	skynet	12	3.2	73.33	✓
	trampoline6	12	6.42	46.43	✗
	trampoline9	12	7.4	38.33	✗
Average Runtime Memory Size Reduction:				62.20%	
Adaptation Success Rate:				75%	
Rhizomatic (Max Mem.: 47 MB Mem. Thold: 20 MB)	BTS-msg	47	19.6	58.30	✓
	BTS-msg-gdl	47	19.6	58.30	✓
	BTS-msg-prd	43	18.6	56.74	✓
	msg-api	47	18.6	60.43	✓
	msg-srvc	46	24.42	46.89	✗
Average Runtime Memory Size Reduction:				56.13%	
Adaptation Success Rate:				80%	

To assess ACADIA's effectiveness, we applied the adaptation strategy described in Section 6.4 on the three subject applications while running each of their example scenarios. Each application developer has introduced a set of example scenarios in which the subject application is used for different functionalities. There are 16 example scenarios proposed for Quasar in its documentation [15], 12 example scenarios implemented for Tascalate [20], and 5 example scenarios for Rhizomatic [17]. We executed each example scenario on the adaptive-version of the corresponding application. To instigate adaptation, we assigned a memory threshold in the range of 42%-64% of the maximum runtime memory required for execution of the non-adaptive version of each application. This assignment of memory threshold was intended to be both challenging, meaning that the original non-adaptive version of applications would not be able to execute under such memory limits, and realistic to provide the possibility of the adaptation strategy overcoming the memory limits. Obviously, if the memory threshold is set to extremely low levels, no adaptation strategy can possibly succeed.

For the purpose of module replacement in the adaptation strategy, we prepared memory-efficient alternative implementations for two modules in each application. To that end, we partially modified the implementations of a few packages inside these modules to provide the same functionality with less runtime memory. More specifically, we added frequent invocations of Java garbage collector (`java.lang.System.gc()`) in the source code that makes the JVM recycle unused objects to make the memory they currently occupy available. We also changed the implementation of certain modules to store specific data in the disk instead of RAM. This decision might have the disadvantage of increasing the execution time. However, memory-constrained devices in modern systems often rely on a solid-state drive, which is faster than traditional hard disk drives, effectively mitigating the loss in execution time when reducing runtime memory usage. The adaptation strategy monitored the state of the applications during the execution of each example scenario. Accordingly, it loaded and unloaded a set of applications' modules using ACADIA, depending on the size of the runtime memory and the required modules, to meet the adaptation objective, i.e., reducing the runtime memory usage below the threshold.

Table 6.2 demonstrates the results of this experiment. It includes the application names with their assigned memory threshold and the list of example scenarios in our experiment. For each scenario, we measured the average runtime memory size in Megabytes (MB) throughout the execution of a scenario. For each example scenario, Table 6.2 includes the average runtime memory usage before and after executing the adaptation strategy and whether the adaptation strategy could successfully reduce the runtime memory usage below the predefined memory threshold. As shown in Table 6.2, ACADIA was able to meet the adaptation strategy's objective in 81%, 75%, and 80% of the example scenarios regarding Quasar, Tascalate, and Rhizomatic applications, respectively. In a few cases ACADIA failed to reduce the runtime memory size below the threshold due to the number of running modules at the moment, yet, it was able to reduce the runtime memory usage by the average of about 40% to 62% in the subject applications. These experiments confirm that, for an overwhelming number of scenarios, ACADIA was able to effectively enable the adaptation strategy to unload and re-load applications' modules and modify their dependencies to reduce the average runtime memory usage.

More importantly, the experiments show that the same adaptation logic can be applied to effectively adapt three real-world Java applications without requiring any changes to the manner in which they are implemented. Indeed, the same adaptation logic can be applied to any Java 9+ application. There is nothing particularly unique about the three applications that were selected for our experiments. It is also important to note that whether the adaptation logic succeeds in meeting its objectives depends on the properties of the system. For instance, whether the adaptation logic can actually reduce the memory footprint of a system below a user-defined threshold depends, at least in part, on whether there are replacement modules with varying levels of memory usage. While we created such replacement modules in our experiments to provide a set of adaptation choices, we did not change the manner in which the applications are implemented. That is, all applications used in our experiments are purely Java 9+ applications.

6.5.3 RQ2: ACADIA's Performance

To assess ACADIA's performance, we answer this research question in terms of the execution time of its Static Analyzer and Runtime Controller compartments discussed in Section 6.3. Table 6.3 reports the execution time of the Static Analyzer and the Runtime Controller for each subject application's example scenario noted as Static and Runtime, respectively. The Static compartment takes between 16 to 36 seconds to statically analyze the subject applications. However, it is executed once, and often offline before running an application, and hence, it does not affect the application's runtime. The Runtime compartment's average execution time ranges from 15 to 35 milliseconds for the subject applications. It includes the time ACADIA requires to perform adaptation operations, such as loading/unloading modules, and updating their dynamic architectural model. The results of Table 6.4 indicate that ACADIA's execution time is only a small fraction of the example scenarios' execution time (reported in the "Single" column in Table 6.4), i.e., about 5-8% on average.

Table 6.3: Results for ACADIA's execution time regarding its Static Analyzer and Runtime Controller compartments

Application Name	Example Scenario	ACADIA Exec. Time (ms) Static	Runtime
Quasar	Fiber	24,375	49
	FiberAsync		47
	IO		40
	Channel		41
	Actor		35
	SrvrBhvr		26
	ProxySrvr		31
	EventSrc		34
	Supervisor		30
	ReactStrms		35
	Issues		33
	Migration		27
	PingPong		37
	GenEvent		33
	GenServer		33
	GalaxySrvr		31
Average:			35.12
Tascalate	cdi-owb	35,721	32
	cdi-weld3		25
	cd-weld4		33
	cglib		33
	common		34
	dyninvoke		40
	jee		24
	lambdas		32
	retro-lmbds		34
	skynet		35
	trampoline6		29
	trampoline9		39
Average:			32.5
Rhizomatic	BTS-msg	16,946	14
	BTS-msg-gdl		10
	BTS-msg-prd		12
	msg-api		9
	msg-service		30
Average:			15

6.5.4 RQ3: Overhead on Applications' Execution

As described in Section 6.3, ACADIA defines and creates module *Wrappers* implemented using `java.lang.ModuleLayer` to manage, load, and unload an application's modules. Such an implementation imposes an overhead in terms of an application's execution time. However, since the defined module wrappers only represent a group of modules with a specific class loader to enable loading and unloading them at runtime, they are very efficient.

To answer RQ3, we measured the average execution time of the example scenarios both when all modules are loaded into a single module wrapper (i.e., without the overhead of wrappers since any Java 9+ system must exist in at least a single `java.lang.ModuleLayer`, which are used to implement a module wrapper) and when the modules are loaded into separate module wrappers (i.e., with the overhead of wrappers). Table 6.4 demonstrates the results of this study. The table includes the execution time of each example scenario executing in a *Single* wrapper and after splitting modules into *Separate* wrappers. The last column reports the percentage of the overhead caused by module wrappers regarding the applications' execution time. As shown in Table 6.4, ACADIA imposes an overhead ranging from 4.3% to 5.4% of the applications' execution time, on average, due to exploiting module wrappers. This overhead is similar to that of adaptation techniques and frameworks in the literature, e.g., 5-10% in Rainbow [83].

Overall, the results of our evaluation indicate that ACADIA can effectively enable architecture-based adaptation capabilities in Java 9+ applications without requiring any change to the applications' implementation, with a low overhead in terms of execution time.

6.6 Related Work

This section provides an overview of the prior work on dynamic software adaptation and architecture-based software adaptation.

Dynamic Adaptation: Dynamic adaptability has become one of the major properties of large software

Table 6.4: Results for the overhead on the execution time of the subject apps

Application Name	Example Scenario	AVG Exec. Time (ms) Single	Separated	Overhead%
Quasar	Fiber	278	313	12.59%
	FiberAsync	272	280	2.94 %
	IO	254	265	4.33 %
	Channel	2057	2079	1.07%
	Actor	576	604	4.86 %
	ServerBehavior	444	479	7.88 %
	ProxyServer	153	167	9.15 %
	EventSource	335	338	0.90 %
	Supervisor	342	355	3.80 %
	ReactiveStreams	234	240	2.56 %
	Issues	1280	1367	6.80 %
	Migration	1237	1306	5.58 %
	PingPong	1218	1254	2.96 %
	GenEvent	1223	1265	3.43 %
	GenServer	1217	1293	6.24 %
	GalaxyServer	1234	1286	4.21 %
Average:				4.96%
Tascalate	cdi-owb	613	638	4.08 %
	cdi-weld3	632	648	2.53 %
	cd-weld4	616	671	8.93 %
	cglib	615	623	1.30 %
	common	613	616	0.49 %
	dyninvoke	612	626	2.29 %
	jee	620	634	2.26 %
	lambdas	611	643	5.24 %
	retro-lambdas	615	652	6.02 %
	skynet	613	648	5.71 %
	trampoline6	612	671	9.64 %
	trampoline9	619	641	3.55 %
Average:				4.34 %
Rhizomatic	BTS-msg	171	180	5.26 %
	BTS-msg-gradle	167	171	2.40 %
	BTS-msg-prod	178	192	7.87 %
	msg-api	194	211	8.76 %
	msg-service	208	213	2.40%
Average:				5.34 %

applications in many different domains [111]. Software engineering community has proposed various techniques for dynamic adaptation of software systems. Some address dynamic adaptation by providing models to specify dynamically adaptive software systems as well as tools for implementing such systems [64, 70, 166, 62, 108]. More specifically, they introduce models to specify and execute adaptive systems [70, 166, 62] or propose a methodology for modeling and validation of dynamic adaptation [108]. Moreover, some studies focus on dynamic adaptation in service-based applications [99] by providing a runtime support [167] or a framework to specify and analyze dynamically configurable service architectures [110]. iPOJO [107] proposes a service-oriented component framework for handling dynamic adaptation. Klus et al. [138] present a dynamic adaptive system infrastructure and an underlying component model.

Some studies have addressed dynamic adaptation from other perspectives, e.g., in multi-cloud or distributed systems [87, 211], investigating safe adaptation [80], utilizing dynamic product line architecture [129, 102, 118], using software process technology [199], and addressing dynamic adaptation from a security perspective [61].

Architecture-Based Adaptation: As software system's architecture directly impacts its quality attributes, the literature has studied a range of architectural styles with regards to their support for structural and behavioral changes at runtime [196, 171], e.g., presenting an architectural strategy for self-adapting systems [209] or a reference architecture for system configuration and behavior adaptation [73].

Plenty of approaches have leveraged architectural models to address dynamic adaptation of software systems [84, 68, 91, 103]. Particularly, Cheng et al. [84] utilize externalized architectural models for adaptation of pervasive computing systems based on performance-related criteria. Bencomo et al. [68] use architectural models for representation, generation, and operation of highly configurable component-based systems. Additionally, Cu et al. [91] leverage an architectural variability mechanism to present an approach that automatically updates both source code and running code during the system evolution. However, due to the uncertainty of many existing models, D'Ippolito et al. [103] propose a tiered framework for combining specific models with different assumptions and risks.

Several research groups have previously developed frameworks for realization of architecture-based adaptation, such as Rainbow [115], c2.fw [94], ArchJava [58], Prism-MW [154], and ActivFORMS [132]. For instance, Rainbow [115] uses architecture models and styles to provide a reusable infrastructure that monitors a running software system and updates its architectural model based on different self-adaptation strategies. Moreover, Swanson et al. present Refract that extends Rainbow with new components and algorithms targeting failure avoidance [191].

Prior studies have investigated architecture-based adaptation in other contexts, e.g., service-based applications [66, 76, 81, 67], distributed systems [212, 75], microservices [109], internet-of-things (IoT) [210], and mobile computing [125]. More specificall, Weyns et al. [212] present an architectural style for designing decentralized self-adaptive systems. Similar to ACADIA, Florio et al. [109] introduce Gru, an approach that adds adaptation capabilities to microservices without changing their implementations. In the context of IoT, Weyns et al. [210] propose an architecture-based adaptation approach for automating the management of IoT systems. Hallsteinsen et al. [125] introduce MADAM, a generic middleware for adaptation in mobile applications.

While the above-mentioned techniques have provided the inspiration for our work, all require a software system to be designed and developed for adaptation using the corresponding frameworks. None of the above-mentioned solutions are capable of readily adapting applications built using one of the most popular programming languages without requiring any changes to the manner in which the applications are built. ACADIA does not require any changes to an implemented system other than using the standard, albeit relatively recently introduced, Java constructs, enabling automatic architecture-based adaptation support for and evaluation on many large, real-world Java applications, which are abundantly available on open-source software repositories (e.g., GitHub).

6.7 Discussion

The capability to retain an accurate representation of a software system's architecture at runtime and make adaptation decisions in terms of its architectural characteristics, e.g., components and interfaces, is referred to as architecture-based adaptability. Despite its potential, architecture-based adaptability has not gained traction in industry, as it often complicates the development of software by requiring non-trivial modification to the manner it is developed. This chapter introduces ACADIA, a framework that leverages JPMS and static analysis techniques to automatically enable architecture-based adaptation support in any Java 9+ application. Using ACADIA, an engineer can focus on the development of adaptation logic, which can obtain the dynamic architecture of the running Java system and apply changes to it through an API. Our evaluation results indicate that ACADIA can effectively maintain and modify the architecture of any Java 9+ application at runtime with very low overhead and requiring no changes to its implementation.

Chapter 7

Conclusion

A software system's architecture comprises the principal design decisions employed in the system's construction, and it is often conceptualized in terms of high-level constructs, such as software components, connectors, and their interfaces. On the other hand, programming languages usually provide low-level constructs, such as classes, methods, and variables. Bridging this gap, yet, remains a challenge as a result of which the software engineering research community has previously advocated for architecture-based development.

Accordingly, modern programming languages and frameworks have recently provided extensive support for architecture-based development by incorporating the notion of software components or modules representing an architectural construct, e.g., modules in Java 9+ or C++20 and dynamic feature modules in Android. The support of modularization in software development helps developers with improved software quality, easier maintainability, and better scalability. Despite all the benefits, it introduces new challenges in different phases of development and execution of software application, including build-time, module installation-time, and run-time.

At build-time, developers explicitly specify the system module declarations and their dependencies which might be inconsistent with the module dependencies implemented in the system. These architectural inconsistencies can lead to severe security, maintainability, and scalability consequences. Furthermore when installing a module during the system's run-time, an installation request may fail due to various contextual reasons. The failure at installing a module can result in run-time exceptions or unexpected behavior of the applications. Finally, the application modules can be loaded and unloaded at run-time. Loading and unloading modules change the system's architecture by defining new dependencies. As a result, it can introduce new architectural inconsistencies to the running system.

This dissertation presents validation and verification approaches to address the challenges of modular software development in different phases of software development and execution. Section 7.1 describes the research contributions in detail.

7.1 Research Contributions

This dissertation makes the following contributions to the Software Engineering research community:

- *A comprehensive defect model of inconsistent module dependencies in Java 9+ applications*: I defined a novel defect model consisting of 8 types of architectural inconsistencies in Java 9+ applications that may arise while specifying module dependencies using Java Platform Module System (JPMS). The inconsistent module dependencies in this defect model are in fact unnecessary specified dependencies in the module description files, e.g., where a module exposes more of its internals than are used or requires internals of other modules that it never uses.
- *Automated detection and repair of modular inconsistencies in Java 9+ applications*: I proposed a novel approach that leverages the defect model of inconsistent module dependencies and static analyses to automatically detect and repair the identified architectural inconsistencies within Java 9+ applications both statically at build-time and dynamically at run-time. This novel technique detects modular inconsistencies with 100% accuracy and robustly repairs them without introducing any unexpected behavior or disruptions to the applications' execution.
- *A comprehensive defect model of dynamic delivery faults in Android app bundles*: I constructed a novel defect model in terms of *Failed State Contexts (FSC)*, i.e., a context or situation in which a dynamic feature module installation request fails. This defect model consists of 11 types of FSCs based on the

root cause of the failure, each depending on different contextual reasons.

- *Automated dynamic delivery testing in Android*: I proposed a novel test augmentation approach for testing dynamic delivery in Android app bundles. The proposed approach considers the dynamic feature modules' installation life-cycle and their fault models to effectively test the behavior of Android applications during the module installation. It leverages static and dynamic analysis techniques to (1) identify tests that reach dynamic feature modules installation requests and (2) modify the identified tests with injecting a combination of GUI and system events to generate new tests that induce different conditions, which may cause an installation request to fail.

- Tools and Experiments:
  - *Automatic detection and repair of architectural inconsistencies in Java*: I implemented the proposed technique and empirically evaluated it on 52 Java 9+ applications with appropriate metrics to demonstrate its effectiveness and efficiency. Furthermore, the developed tool is publicly released [52] to help other researchers reuse and extend the proposed approach and build more advanced techniques on top of them.
  - *Dynamic delivery testing in Android*: I implemented the proposed technique and empirically evaluated it on 25 Android applications with appropriate metrics to demonstrate its effectiveness and efficiency. Furthermore, the developed tool is publicly released [41] to help other researchers reuse and extend the proposed approach and build more advanced techniques on top of them.

- Dataset
  - A dataset of 567 inconsistent module dependency defects in real-world Java applications: This data set contains the source code of 38 Java 9+ applications with architectural inconsistencies specified in their module description files. The dataset contains additional data, such as the type of the architectural inconsistencies, the affected modules, and packages for each application to help other researcher use the dataset easily. This dataset is available upon request.
  - A dataset of 204 reproducible dynamic delivery defects in real-world Android applications: This dataset contains the source code and apk files of 25 Android applications with crashes or unexpected behavior during the dynamic feature module installation failures. 75 defects among

this list was confirmed by the application developers. The dataset also includes some additional data, such as the type of dynamic delivery defect, the affected dynamic feature modules, and the applications defective behavior to help other researcher use the dataset easily. This dataset is available upon request.

7.2 Future Directions

Automatic modularization of Java applications with no architectural inconsistencies. Migrating non-modular Java applications, before Java 9, to modular Java 9+ application requires significant engineering effort and is highly time-consuming. Therefore, many developers do not modularize their Java applications to use Java Platform Module System (JPMS) or migrate them with inadequate modularization and many unnecessary module dependencies, introducing numerous architectural inconsistencies. For instance, developers might unnecessarily expose the applications' internals or require the internals of other modules or JDK modules, which seriously impair the applications' security, maintainability, and scalability. To that end, the software engineering community needs to address the issues of modularization of Java applications and migrating them to Java 9+. More specifically, researchers need to develop automated approaches that consider related security, encapsulation, and scalability metrics and efficiently modularize Java applications with no architectural inconsistencies.

Automatic repair of dynamic delivery defects in Android applications. Dynamic delivery defects in Android can result in critical failures, i.e., crashing or unexpected behaviors of applications. Therefore, it requires developers' extra attention to localize and fix them. This dissertation helps developers with detecting the issues and testing them. However, developers need to manually repair the identified dynamic delivery defects, as there is no automated technique to resolve such failures. The software engineering research community should focus on resolving dynamic delivery failures in Android applications and devise an automated and efficient approach that robustly repair such failures in Android application to advance this line of research line further.

Bibliography

[1] Beautiful Soup. https://www.crummy.com/software/BeautifulSoup/.

[2] Oracle Corporation. API specification for the Java Platform, Standard Edition: Class ServiceLoader. https://docs.oracle.com/javase/7/docs/api/java/util/ServiceLoader.html.

[3] OSGI Alliance. https://www.osgi.org/developer/specifications/.

[4] puniverse/quasar. https://github.com/puniverse/quasar, 2013. Accessed: 2022-01-10.

[5] Google guice. https://github.com/google/guice, 2014. Accessed: 2022-01-10.

[6] The java platform module system (jsr 376). https://cr.openjdk.java.net/~mr/jigsaw/spec/, 2017. Accessed: 2022-01-10.

[7] Project Jigsaw. http://openjdk.java.net/projects/jigsaw/, 2017.

[8] Project jigsaw. https://openjdk.java.net/projects/jigsaw/, 2017. Accessed: 2022-01-10.

[9] Tascalate JavaFlow. https://github.com/vsilaev/tascalate-javaflow, 2017. Accessed: 2022-01-10.

[10] ANTLR. http://www.antlr.org, 2018.

[11] GitHub. https://github.com, 2018.

[12] GitHub googlearchive Issues 1. https://github.com/googlearchive/android-dynamic-features/issues/1, 2018.

[13] GitHub googlearchive Issues 2. https://github.com/googlearchive/android-dynamic-features/issues/2, 2018.

[14] Quasar. http://docs.paralleluniverse.co/quasar/, 2018. Accessed: 2022-01-10.

[15] Quasar examples. http://docs.paralleluniverse.co/quasar/#examples, 2018. Accessed: 2022-01-10.

[16] Rhizomatic. https://github.com/Rhizomatic-IO/rhizomatic, 2018. Accessed: 2022-01-10.

[17] Rhizomatic Samples. https://github.com/Rhizomatic-IO/rhizomatic-samples, 2018. Accessed: 2022-01-10.

[18] ServiceLoader (Java SE 9 & JDK 9). https://docs.oracle.com/javase/9/docs/api/java/util/ServiceLoader.html, 2018.

[19] Soot GitHub Issue. https://github.com/Sable/soot/issues, 2018.

[20] Tascalate JavaFlow Examples. https://github.com/vsilaev/tascalate-javaflow-examples, 2018. Accessed: 2022-01-10.

[21] The State of the Octoverse 2017. https://octoverse.github.com/, 2018.

[22] TIOBE Index for August 2018. https://www.tiobe.com/tiobe-index/, 2018.

[23] GitHub googlearchive Issues 36. https://github.com/googlearchive/android-dynamic-features/issues/36, 2019.

[24] GitHub googlearchive Issues 41. https://github.com/googlearchive/android-dynamic-features/issues/41, 2019.

[25] Network exceptions in android. https://developer.android.com/guide/topics/connectivity/cronet/reference/org/chromium/net/NetworkException, 2019.

[26] Stackoverflow question 56255600. https://stackoverflow.com/questions/56255600, 2019.

[27] Stackoverflow question 56454470. https://stackoverflow.com/questions/56454470, 2019.

[28] Stackoverflow question 58611149. https://stackoverflow.com/questions/58611149, 2019.

[29] About android app bundles. https://developer.android.com/guide/app-bundle, 2020.

[30] Android developers blog: Recent android app bundle improvements and timeline for new apps on google play. https://android-developers.googleblog.com/2020/08/recent-android-app-bundle-improvements.html, 2020.

[31] k-9 mail app. https://github.com/EverlastingHopeX/K-9-Modularization, 2020.

[32] Mobile Operating System Market Share Worldwide. https://gs.statcounter.com/os-market-share/mobile/worldwide/#monthly-201901-201912, 2020.

[33] Modulelayer (java se 15 jdk 15). https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/lang/ModuleLayer.html, 2020. Accessed: 2022-01-10.

[34] Stackoverflow question 60181875. https://stackoverflow.com/questions/60181875/, 2020.

[35] Stackoverflow question 60328502. https://stackoverflow.com/questions/60328502, 2020.

[36] Stackoverflow question 61384372. https://stackoverflow.com/questions/61384372, 2020.

[37] Stackoverflow question 63100883. https://stackoverflow.com/questions/63100883, 2020.

[38] Storage exceptions in android. https://developers.google.com/android/reference/com/google/firebase/storage/StorageException, 2020.

[39] Ui/application exerciser monkey. https://developer.android.com/studio/test/monkey, 2020.

[40] Appium mobile app testing tool. http://appium.io/, 2021.

[41] Deltadroid. https://sites.google.com/view/deltadroid/home, 2021.

[42] Dot language. https://graphviz.org/doc/info/lang.html, 2021. Accessed: 2022-01-10.

[43] Exemplars - Self-Adaptive Systems Artifacts and Model Problems. https://www.hpi.uni-potsdam.de/giese/public/selfadapt/exemplars/, 2021. Accessed: 2022-01-20.

[44] Gradle build tool. https://gradle.org/, 2021.

[45] Play core library. https://developer.android.com/guide/playcore, 2021.

[46] Play feature delivery. https://developer.android.com/guide/playcore/dynamic-delivery, 2021.

[47] Play feature delivery - handle request errors. https://developer.android.com/guide/playcore/feature-delivery/on-demand#handle_request_errors, 2021.

[48] StackOverFlow. https://stackoverflow.com, 2021.

[49] Ux best practices for on demand delivery. https://developer.android.com/guide/playcore/feature-delivery/ux-guidelines#communicate, 2021.

[50] App crawler. https://developer.android.com/studio/test/other-testing-tools/app-crawler, 2022.

[51] . https://sites.google.com/view/archadaptacadia, 2022.

[52] Darcy. https://sites.google.com/view/darcy-plus-plus/home, 2022.

[53] Thor Issue Repository. https://github.com/cs-au-dk/thor/issues/2, 2022.

[54] TimeMachine Issue Repository. https://github.com/DroidTest/TimeMachine/issues/7, 2022.

[55] M. Abi-Antoun, J. Aldrich, D. Garlan, B. Schmerl, N. Nahas, and T. Tseng. Improving system dependability by enforcing architectural intent. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 1–7. ACM, 2005.

[56] C. Q. Adamsen, G. Mezzetti, and A. Møller. Systematic execution of android test suites in adverse conditions. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 83–93, 2015.

[57] J. Aldrich, C. Chambers, and D. Notkin. Archjava: Connecting software architecture to implementation. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE ’02, pages 187–197, New York, NY, USA, 2002. ACM.

[58] J. Aldrich, C. Chambers, and D. Notkin. Archjava: Connecting software architecture to implementation. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE ’02, page 187–197, New York, NY, USA, 2002. Association for Computing Machinery.

[59] J. Aldrich, C. Omar, A. Potanin, and D. Li. Language-based architectural control. In *Proceedings of the International Workshop on Aliasing, Capabilities and Ownership (IWACO)*, pages 1–11, 2014.

[60] D. Amalfitano, A. R. Fasolino, P. Tramontana, and N. Amatucci. Considering context events in event-based testing of mobile applications. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*, pages 126–133, 2013.

[61] M. Amoud and O. Roudies. Dynamic adaptation and reconfiguration of security in mobile devices. In *2017 International Conference On Cyber Incident Response, Coordination, Containment & Control (Cyber Incident)*, pages 1–6. IEEE, 2017.

[62] M. Amoui, M. Derakhshanmanesh, J. Ebert, and L. Tahvildari. Achieving dynamic adaptation via management and interpretation of runtime models. *Journal of Systems and Software*, 85(12):2720–2737, 2012.

[63] S. Anand, M. Naik, M. J. Harrold, and H. Yang. Automated concolic testing of smartphone apps. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pages 1–11, 2012.

[64] J. Andersson, R. d. Lemos, S. Malek, and D. Weyns. Modeling dimensions of self-adaptive software systems. In *Software engineering for self-adaptive systems*, pages 27–47. Springer, 2009.

[65] T. Apiwattanapong, R. Santelices, P. K. Chittimalli, A. Orso, and M. J. Harrold. Matrix: Maintenance-oriented testing requirements identifier and examiner. In *Testing: Academic & Industrial Conference-Practice And Research Techniques (TAIC PART'06)*, pages 137–146. IEEE, 2006.

[66] L. Baresi, E. Di Nitto, C. Ghezzi, and S. Guinea. A framework for the deployment of adaptable web service compositions. *Service Oriented Computing and Applications*, 1(1):75–91, 2007.

[67] L. Baresi and L. Pasquale. Adaptation goals for adaptive service-oriented architectures. In *Relating Software Requirements and Architectures*, pages 161–181. Springer, 2011.

[68] N. Bencomo, P. Grace, C. Flores, D. Hughes, and G. Blair. Genie. In *2008 ACM/IEEE 30th International Conference on Software Engineering*, pages 811–814. IEEE, 2008.

[69] M. Berlingerio, M. Coscia, F. Giannotti, A. Monreale, and D. Pedreschi. Multidimensional networks: foundations of structural analysis. *World Wide Web*, 16(5-6):567–593, 2013.

[70] T. E. Bihari and K. Schwan. Dynamic adaptation of real-time software. *ACM Transactions on Computer Systems (TOCS)*, 9(2):143–174, 1991.

[71] R. A. Bittencourt and D. D. S. Guerrero. Comparison of graph clustering algorithms for recovering software architecture module views. In *Software Maintenance and Reengineering, 2009. CSMR'09. 13th European Conference on*, pages 251–254. IEEE, 2009.

[72] E. Bouwers, A. van Deursen, and J. Visser. Quantifying the encapsulation of implemented software architectures. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, pages 211–220. IEEE, 2014.

[73] V. Braberman, N. D'Ippolito, J. Kramer, D. Sykes, and S. Uchitel. Morph: A reference architecture for configuration and behaviour self-adaptation. In *Proceedings of the 1st International Workshop on Control Theory for Software Engineering*, pages 9–16, 2015.

[74] L. C. Briand, S. Morasca, and V. R. Basili. Measuring and assessing maintainability at the end of high level design. In *Software Maintenance, 1993. CSM-93, Proceedings., Conference on*, pages 88–87. IEEE, 1993.

[75] A. Brogi, J. Carrasco, J. Cubo, F. D'Andria, E. Di Nitto, M. Guerriero, D. Pérez, E. Pimentel, and J. Soldani. Seaclouds: an open reference architecture for multi-cloud governance. In *European Conference on Software Architecture*, pages 334–338. Springer, 2016.

[76] A. Bucchiarone, C. Cappiello, E. D. Nitto, R. Kazhamiakin, V. Mazza, and M. Pistore. Design for adaptation of service-based applications: Main issues and requirements. In *Service-Oriented Computing. ICSOC/ServiceWave 2009 Workshops*, pages 467–476. Springer, 2009.

[77] J. Buckley, S. Mooney, J. Rosik, and N. Ali. Jittac: a just-in-time tool for architectural consistency. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 1291–1294. IEEE, 2013.

[78] Y. Cao, G. Wu, W. Chen, and J. Wei. Crawldroid: Effective model-based gui testing of android apps. In *Proceedings of the Tenth Asia-Pacific Symposium on Internetware*, pages 1–6, 2018.

[79] M. Caporuscio, H. Muccini, P. Pelliccione, and E. Di Nisio. Rapid system development via product line architecture implementation. In *International Workshop on Rapid Integration of Software Engineering Techniques*, pages 18–33. Springer, 2005.

[80] W. Cazzola, A. Ghoneim, and G. Saake. Software evolution through dynamic adaptation of its oo design. In *Objects, Agents, and Features*, pages 67–80. Springer, 2004.

[81] A. Charfi, T. Dinkelaker, and M. Mezini. A plug-in architecture for self-adaptive web service compositions. In *2009 IEEE International Conference on Web Services*, pages 35–42. IEEE, 2009.

[82] B. H. C. Cheng, R. de Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic, G. Di Marzo Serugendo, S. Dustdar, A. Finkelstein, C. Gacek, K. Geihs, V. Grassi, G. Karsai, H. M. Kienle, J. Kramer, M. Litoiu, S. Malek, R. Mirandola, H. A. Müller, S. Park, M. Shaw, M. Tichy, M. Tivoli, D. Weyns, and J. Whittle. *Software Engineering for Self-Adaptive Systems: A Research Roadmap*, pages 1–26. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[83] S.-W. Cheng, D. Garlan, and B. Schmerl. Evaluating the effectiveness of the rainbow self-adaptive system. In *2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, pages 132–141. IEEE, 2009.

[84] S.-W. Cheng, D. Garlan, B. Schmerl, J. P. Sousa, B. Spitznagel, P. Steenkiste, and N. Hu. Software architecture-based adaptation for pervasive systems. In *International Conference on Architecture of Computing Systems*, pages 67–82. Springer, 2002.

[85] S. R. Choudhary, A. Gorla, and A. Orso. Automated test input generation for android: Are we there yet? (e). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 429–440, 2015.

[86] A. Christl, R. Koschke, and M.-A. Storey. Equipping the reflexion method with automated clustering. In *Reverse Engineering, 12th Working Conference on*, pages 10–pp. IEEE, 2005.

[87] L. Cianciaruso, F. Di Forenza, E. Di Nitto, M. Miglierina, N. Ferry, and A. Solberg. Using models at runtime to support adaptable monitoring of multi-clouds applications. In *2014 16th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pages 401–408. IEEE, 2014.

[88] E. Constantinou, G. Kakarontzas, and I. Stamelos. Open source software: How can design metrics facilitate architecture recovery? *arXiv preprint arXiv:1110.1992*, 2011.

[89] E. Constantinou, G. Kakarontzas, and I. Stamelos. Towards open source software system architecture recovery using design metrics. In *2011 15th Panhellenic Conference on Informatics*, pages 166–170, Sept 2011.

[90] E. Constantinou, G. Kakarontzas, and I. Stamelos. An automated approach for noise identification to assist software architecture recovery techniques. *Journal of Systems and Software*, 107:142–157, 2015.

[91] C. Cu, R. Culver, and Y. Zheng. Dynamic architecture-implementation mapping for architecture-based runtime software adaptation. In *SEKE*, pages 135–140, 2020.

[92] F. Y. B. Daragh and S. Malek. Deep gui: Black-box gui input generation with deep learning. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 905–916. IEEE, 2021.

[93] E. M. Dashofy, A. v. d. Hoek, and R. N. Taylor. A comprehensive approach for the development of modular software architecture description languages. *ACM Trans. Softw. Eng. Methodol.*, 14(2):199–245, Apr. 2005.

[94] E. M. Dashofy, A. Van der Hoek, and R. N. Taylor. Towards architecture-based self-healing systems. In *Proceedings of the first workshop on Self-healing systems*, pages 21–26, 2002.

[95] T. B. de Assis, A. A. Menegassi, and A. T. Endo. Amplifying tests for cross-platform apps through test patterns. In *SEKE*, pages 55–74, 2019.

[96] P. Deitel. Understanding Java 9 Modules. `https://www.oracle.com/corporate/features/understanding-java-9-modules.html`, 2017.

[97] P. J. Deitel and H. M. Deitel. *Java 9 for Programmers*. Prentice Hall, 2017.

[98] D. Di Nardo, F. Pastore, and L. Briand. Augmenting field data for testing systems subject to incremental requirements changes. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 26(1):1–40, 2017.

[99] E. Di Nitto, C. Ghezzi, A. Metzger, M. Papazoglou, and K. Pohl. A journey to highly dynamic, self-adaptive service-based applications. *Automated Software Engineering*, 15(3):313–341, 2008.

[100] J. A. Diaz-Pace, J. P. Carlino, M. Blech, A. Soria, and M. R. Campo. Assisting the synchronization of ucm-based architectural documentation with implementation. In *2009 Joint Working IEEE/IFIP Conference on Software Architecture (WICSA) & 3rd European Conference on Software Architecture (ECSA)*, pages 151–160. IEEE, 2009.

[101] Z. Ding. *Towards the Use of the Readily Available Tests from the Release Pipeline as Performance Tests. Are We There Yet?* PhD thesis, Concordia University, 2019.

[102] T. Dinkelaker, R. Mitschke, K. Fetzer, and M. Mezini. A dynamic software product line approach using aspect models at runtime. In *5th Domain-Specific Aspect Languages Workshop*, 2010.

[103] N. D'Ippolito, V. Braberman, J. Kramer, J. Magee, D. Sykes, and S. Uchitel. Hope for the best, prepare for the worst: multi-tier control for adaptive systems. In *Proceedings of the 36th International Conference on Software Engineering*, pages 688–699, 2014.

[104] Z. Dong, M. Böhme, L. Cojocaru, and A. Roychoudhury. Time-travel testing of android apps. In *Proceedings of the 42nd International Conference on Software Engineering (ICSE'20)*, pages 1–12, 2020.

[105] S. Ducasse and D. Pollet. Software architecture reconstruction: A process-oriented taxonomy. *IEEE Transactions on Software Engineering*, 35(4):573–591, 2009.

[106] F.-J. Elmer. Classycle: Analysing Tools for Java Class and Package Dependencies. *How Classycle works*, 2012.

[107] C. Escoffier and R. S. Hall. Dynamically adaptable applications with ipojo service components. In *International conference on Software composition*, pages 113–128. Springer, 2007.

[108] F. Fleurey, V. Dehlen, N. Bencomo, B. Morin, and J.-M. Jézéquel. Modeling and validating dynamic adaptation. In *International Conference on Model Driven Engineering Languages and Systems*, pages 97–108. Springer, 2008.

[109] L. Florio and E. Di Nitto. Gru: An approach to introduce decentralized autonomic behavior in microservices architectures. In *2016 IEEE International Conference on Autonomic Computing (ICAC)*, pages 357–362. IEEE, 2016.

[110] H. Foster, A. Mukhija, D. S. Rosenblum, and S. Uchitel. Specification and analysis of dynamically-reconfigurable service architectures. In *Rigorous software engineering for service-oriented systems*, pages 428–446. Springer, 2011.

[111] J. Fox and S. Clarke. Exploring approaches to dynamic adaptation. In *Proceedings of the 3rd International DiscCoTec Workshop on Middleware-Application Interaction*, pages 19–24, 2009.

[112] E. R. Gansner and S. C. North. An open graph visualization system and its applications to software engineering. *SOFTWARE - PRACTICE AND EXPERIENCE*, 30(11):1203–1233, 2000.

[113] J. Garcia, M. Hammad, and S. Malek. Lightweight, obfuscation-resilient detection and family identification of android malware. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 26(3):11, 2018.

[114] J. Garcia, I. Ivkovic, and N. Medvidovic. A comparative analysis of software architecture recovery techniques. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, pages 486–496. IEEE Press, 2013.

[115] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54, 2004.

[116] N. Ghorbani, J. Garcia, and S. Malek. Detection and repair of architectural inconsistencies in java. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 560–571. IEEE, 2019.

[117] M. W. Godfrey and E. H. Lee. Secrets from the monster: Extracting mozilla's software architecture. In *Proceedings of Second Symposium on Constructing Software Engineering Tools (CoSET'00)*. Citeseer, 2000.

[118] H. Gomaa and K. Hashimoto. Dynamic software adaptation for service-oriented product lines. In *Proceedings of the 15th International Software Product Line Conference, Volume 2*, pages 1–8, 2011.

[119] J. Gosling, B. Joy, G. Steele, G. Bracha, A. Buckley, and D. Smith. The Java® Language Specification Java SE 9 Edition. Technical report, Technical report, Oracle Corporation. https:https://docs.oracle.com/javase/specs/jls/se9/html/index.html, 2017.

[120] M. Grechanik, Q. Xie, and C. Fu. Maintaining and evolving gui-directed test scripts. In *2009 IEEE 31st International Conference on Software Engineering*, pages 408–418. IEEE, 2009.

[121] J. Grundy. Multi-perspective specification, design and implementation of software components using aspects. *International Journal of Software Engineering and Knowledge Engineering*, 10(06):713–734, 2000.

[122] T. Gu, C. Cao, T. Liu, C. Sun, J. Deng, X. Ma, and J. Lü. Aimdroid: Activity-insulated multi-level automated testing for android applications. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 103–114. IEEE, 2017.

[123] T. Gu, C. Sun, X. Ma, C. Cao, C. Xu, Y. Yao, Q. Zhang, J. Lu, and Z. Su. Practical gui testing of android applications via model abstraction and refinement. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 269–280. IEEE, 2019.

[124] A. Gurgel, I. Macia, A. Garcia, A. von Staa, M. Mezini, M. Eichberg, and R. Mitschke. Blending and reusing rules for architectural degradation prevention. In *Proceedings of the 13th international conference on Modularity*, pages 61–72. ACM, 2014.

[125] S. Hallsteinsen, J. Floch, and E. Stav. A middleware centric approach to building self-adapting systems. In *International Workshop on Software Engineering and Middleware*, pages 107–122. Springer, 2004.

[126] M. M. Hammad, I. Abueisa, and S. Malek. Tool-assisted componentization of java applications. In *2022 IEEE 19th International Conference on Software Architecture (ICSA)*, pages 36–46. IEEE, 2022.

[127] M. Harman and N. Alshahwan. Automated session data repair for web application regression testing. In *2008 1st International Conference on Software Testing, Verification, and Validation*, pages 298–307. IEEE, 2008.

[128] R. Hay, O. Tripp, and M. Pistoia. Dynamic detection of inter-application communication vulnerabilities in android. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 118–128, 2015.

[129] A. Helleboogh, D. Weyns, K. Schmid, T. Holvoet, K. Schelfthout, and W. Van Betsbrugge. Adding variants on-the-fly: Modeling meta-variability in dynamic software product lines. In *Proceedings of the Third International Workshop on Dynamic Software Product Lines (DSPL@ SPLC 2009)*, pages 18–27. Carnegie Mellon University; Pittsburgh, PA, USA, 2009.

[130] L. Hendren. Uses of the Soot Framework. `http://www.sable.mcgill.ca/~hendren/sootusers/`, 2018.

[131] G. Hu, L. Zhu, and J. Yang. Appflow: using machine learning to synthesize robust, reusable ui tests. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 269–282, 2018.

[132] M. U. Iftikhar and D. Weyns. Activforms: A runtime environment for architecture-based adaptation with guarantees. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, pages 278–281. IEEE, 2017.

[133] I. Ivkovic and M. Godfrey. Enhancing domain-specific software architecture recovery. In *Program Comprehension, 2003. 11th IEEE International Workshop on*, pages 266–273. IEEE, 2003.

[134] R. Jabbarvand, J.-W. Lin, and S. Malek. Search-based energy testing of android. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 1119–1130. IEEE, 2019.

[135] G. S. G. B. A. B. D. S. James Gosling, Bill Joy. The java language specification, java se 9 edition. `https://docs.oracle.com/javase/specs/jls/se9/html/index.html`, 2017. Accessed: 2022-01-10.

[136] C. S. Jensen, M. R. Prasad, and A. Møller. Automated testing with targeted event sequence generation. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 67–77, 2013.

[137] Y. Kim, Z. Zu, M. Kim, M. B. Cohen, and G. Rothermel. Hybrid directed test suite augmentation: An interleaving framework. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, pages 263–272. IEEE, 2014.

[138] H. Klus, D. Niebuhr, and A. Rausch. A component model for dynamic adaptive systems. In *International workshop on Engineering of software services for pervasive environments: in conjunction with the 6th ESEC/FSE joint meeting*, pages 21–28, 2007.

[139] Y. Koroglu and A. Sen. Tcm: test case mutation to improve crash detection in android. In *International Conference on Fundamental Approaches to Software Engineering*, pages 264–280. Springer, 2018.

[140] R. Koschke. Architecture reconstruction. In *Software Engineering*, pages 140–173. Springer, 2006.

[141] R. Koschke, P. Frenzel, A. P. Breu, and K. Angstmann. Extending the reflexion method for consolidating software variants into product lines. *Software Quality Journal*, 17(4):331–366, 2009.

[142] R. Koschke and D. Simon. Hierarchical reflexion models. In *null*, page 36. IEEE, 2003.

[143] J. Lakos. Large-scale c++ software design. *Reading, MA*, 173:217–271, 1996.

[144] P. Lam, E. Bodden, O. Lhoták, and L. Hendren. The soot framework for java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, volume 15, page 35, 2011.

[145] L. Lambers, D. Strüber, G. Taentzer, K. Born, and J. Huebert. Multi-granular conflict and dependency analysis in software engineering based on graph transformation. In *Proceedings of the 40th International Conference on Software Engineering*, pages 716–727, 2018.

[146] K. Lau and Z. Wang. Software component models. *IEEE Transactions on Software Engineering*, 33(10):709–724, Oct 2007.

[147] D. M. Le, P. Behnamghader, J. Garcia, D. Link, A. Shahbazian, and N. Medvidovic. An empirical study of architectural change in open-source software systems. In *Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conference on*, pages 235–245. IEEE, 2015.

[148] Z. Li, Y. Wang, Z. Lin, S.-C. Cheung, and J.-G. Lou. Nufix: Escape from nuget dependency maze. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, pages 1545–1557. IEEE, 2022.

[149] M. Linares-Vásquez, M. White, C. Bernal-Cárdenas, K. Moran, and D. Poshyvanyk. Mining android app usages for generating actionable gui-based execution scenarios. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 111–122. IEEE, 2015.

[150] T. Lutellier, D. Chollak, J. Garcia, L. Tan, D. Rayside, N. Medvidović, and R. Kroeger. Comparing software architecture recovery techniques using accurate dependencies. In *Proceedings of the 37th International Conference on Software Engineering-Volume 2*, pages 69–78. IEEE Press, 2015.

[151] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 224–234, 2013.

[152] R. Mahmood, N. Mirzaei, and S. Malek. Evodroid: Segmented evolutionary testing of android apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 599–609, 2014.

[153] S. Mak and P. Bakker. *Java 9 Modularity: Patterns and Practices for Developing Maintainable Applications.* " O'Reilly Media, Inc.", 2017.

[154] S. Malek, M. Mikic-Rakic, and N. Medvidovic. A style-aware architectural middleware for resource-constrained, distributed systems. *IEEE Transactions on Software Engineering*, 31(3):256–272, 2005.

[155] P. Manadhata and J. M. Wing. Measuring a system's attack surface. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA SCHOOL OF COMPUTER SCIENCE, 2004.

[156] P. K. Manadhata, K. M. Tan, R. A. Maxion, and J. M. Wing. An approach to measuring a system's attack surface. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA SCHOOL OF COMPUTER SCIENCE, 2007.

[157] P. K. Manadhata and J. M. Wing. An attack surface metric. *IEEE Transactions on Software Engineering*, (3):371–386, 2010.

[158] K. Mao, M. Harman, and Y. Jia. Sapienz: Multi-objective automated testing for android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 94–105, 2016.

[159] K. Mao, M. Harman, and Y. Jia. Crowd intelligence enhances automated mobile testing. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 16–26. IEEE, 2017.

[160] O. Maqbool and H. Babri. Hierarchical clustering for software architecture recovery. *IEEE Transactions on Software Engineering*, 33(11), 2007.

[161] N. Medvidovic, A. Egyed, and P. Gruenbacher. Stemming architectural erosion by coupling architectural discovery and recovery. In *STRAW*, volume 3, pages 61–68, 2003.

[162] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, Jan 2000.

[163] A. M. Memon. Automatically repairing event sequence-based gui test suites for regression testing. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 18(2):1–36, 2008.

[164] A. Milani Fard, M. Mirzaaghaei, and A. Mesbah. Leveraging existing tests in automated test generation for web applications. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 67–78, 2014.

[165] M. Mirakhorli, Y. Shin, J. Cleland-Huang, and M. Cinar. A tactic-centric approach for automating traceability of quality concerns. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 639–649. IEEE, 2012.

[166] B. Morin, O. Barais, J.-M. Jézéquel, F. Fleurey, and A. Solberg. Models@ run. time to support dynamic adaptation. *Computer*, 42(10):44–51, 2009.

[167] A. Mukhija, D. S. Rosenblum, H. Foster, and S. Uchitel. Runtime support for dynamic and adaptive service composition. In *Rigorous software engineering for service-oriented systems*, pages 585–603. Springer, 2011.

[168] G. C. Murphy, D. Notkin, and K. J. Sullivan. Software reflexion models: Bridging the gap between design and implementation. *IEEE Transactions on Software Engineering*, 27(4):364–380, 2001.

[169] L. O'Brien, D. Smith, and G. Lewis. Supporting migration to services using software architecture reconstruction. In *Software Technology and Engineering Practice, 2005. 13th IEEE International Workshop on*, pages 81–91. IEEE, 2005.

[170] P. Oreizy, N. Medvidovic, and R. N. Taylor. Runtime software adaptation: framework, approaches, and styles. In *Companion of the 30th international conference on Software engineering*, pages 899–910, 2008.

[171] P. Oreizy and R. N. Taylor. On the role of software architectures in runtime system reconfiguration. *IEE Proceedings-Software*, 145(5):137–145, 1998.

[172] A. C. Paiva, A. Restivo, and S. Almeida. Test case generation based on mutations over user execution traces. *Software Quality Journal*, 28(3):1173–1186, 2020.

[173] L. S. Pinto, S. Sinha, and A. Orso. Understanding myths and realities of test-suite evolution. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pages 1–11, 2012.

[174] D. Pradhan, S. Wang, T. Yue, S. Ali, and M. Liaaen. Search-based test case implantation for testing untested configurations. *Information and Software Technology*, 111:22–36, 2019.

[175] D. Qiu, Q. Zhang, and S. Fang. Reconstructing software high-level architecture by clustering weighted directed class graph. *International Journal of Software Engineering and Knowledge Engineering*, 25(04):701–726, 2015.

[176] A. Radjenovic and R. F. Paige. The role of dependency links in ensuring architectural view consistency. In *Seventh Working IEEE/IFIP Conference on Software Architecture*, pages 199–208. IEEE, 2008.

[177] M. Reinhold. JSR 376: Java Platform Module System. Technical report, Technical report, Oracle Corporation. http://cr.openjdk.java.net/ mr/jigsaw/spec/, 2014.

[178] K. Rubinov. *Automatically generating complex test cases from simple ones*. PhD thesis, Università della Svizzera italiana, 2013.

[179] N. Salehnamadi, A. Alshayban, J.-W. Lin, I. Ahmed, S. Branham, and S. Malek. Latte: Use-case and assistive-service driven automated accessibility testing framework for android. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, pages 1–11, 2021.

[180] N. Sangal, E. Jordan, V. Sinha, and D. Jackson. Using dependency models to manage complex software architecture. In *ACM Sigplan Notices*, volume 40, pages 167–176. ACM, 2005.

[181] R. Santelices, P. K. Chittimalli, T. Apiwattanapong, A. Orso, and M. J. Harrold. Test-suite augmentation for evolving software. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 218–227. IEEE, 2008.

[182] M. Sefika, A. Sane, and R. H. Campbell. Monitoring compliance of a software system with its high-level design models. In *Proceedings of the 18th international conference on Software engineering*, pages 387–396. IEEE Computer Society, 1996.

[183] A. Shahbazian, Y. K. Lee, D. Le, Y. Brun, and N. Medvidovic. Recovering architectural design decisions. In *2018 IEEE International Conference on Software Architecture (ICSA)*. IEEE, 2018.

[184] M. R. Shaheen and L. du Bousquet. Quantitative analysis of testability antipatterns on open source java applications. *QAOOSE 2008-Proceedings*, page 21, 2008.

[185] K. Sharan. The module system. In *Java 9 Revealed*, pages 7–30. Springer, 2017.

[186] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik. Abstractions for software architecture and tools to support them. *IEEE transactions on software engineering*, 21(4):314–335, 1995.

[187] M. Shtern and V. Tzerpos. Clustering Methodologies for Software Engineering. *Adv. Soft. Eng.*, 2012:1:1–1:1, Jan. 2012.

[188] H. Song, G. Huang, F. Chauvel, Y. Xiong, Z. Hu, Y. Sun, and H. Mei. Supporting runtime software architecture: A bidirectional-transformation-based approach. *Journal of Systems and Software*, 84(5):711–723, 2011.

[189] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su. Guided, stochastic model-based gui testing of android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 245–256, 2017.

[190] N. Subramanian and L. Chung. Software architecture adaptability: an nfr approach. In *Proceedings of the 4th International Workshop on Principles of Software Evolution*, pages 52–61, 2001.

[191] J. Swanson, M. B. Cohen, M. B. Dwyer, B. J. Garvin, and J. Firestone. Beyond the rainbow: Self-adaptive failure avoidance in configurable systems. In *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*, pages 377–388, 2014.

[192] P. Tarvainen. Adaptability evaluation at software architecture level. *The Open Software Engineering Journal*, 2(1), 2008.

[193] R. Taylor, N. Medvidovic, and E. Dashofy. *Software Architecture: Foundations, Theory and Practice*. John Wiley and Sons, 2009.

[194] R. Taylor, N. Medvidovic, and D. E.M. *Software Architecture: Foundations, Theory, and Practice*. John Wiley & Sons, 2009.

[195] R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead, Jr., and J. E. Robbins. A component- and message-based architectural style for gui software. In *Proceedings of the 17th International Conference on Software Engineering*, ICSE '95, pages 295–304, New York, NY, USA, 1995. ACM.

[196] R. N. Taylor, N. Medvidovic, and P. Oreizy. Architectural styles for runtime software adaptation. In *2009 Joint Working IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture*, pages 171–180. IEEE, 2009.

[197] J. B. Tran, M. W. Godfrey, E. H. Lee, and R. C. Holt. Architectural repair of open source software. In *Program Comprehension, 2000. Proceedings. IWPC 2000. 8th International Workshop on*, pages 48–59. IEEE, 2000.

[198] N. Ubayashi, J. Nomura, and T. Tamai. Archface: a contract place where architectural design and code meet together. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 75–84. ACM, 2010.

[199] G. Valetto, G. Kaiser, and G. S. Kc. A mobile agent approach to process-based dynamic adaptation of complex software systems. In *European Workshop on Software Process Technology*, pages 102–116. Springer, 2001.

[200] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot: A Java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, pages 214–224. IBM Corp., 2010.

[201] A. Van Deursen, C. Hofmeister, R. Koschke, L. Moonen, and C. Riva. Symphony: View-driven software architecture reconstruction. In *Software Architecture, 2004. WICSA 2004. Proceedings. Fourth Working IEEE/IFIP Conference on*, pages 122–132. IEEE, 2004.

[202] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee. The koala component model for consumer electronics software. *Computer*, 33(3):78–85, March 2000.

[203] H. Wang, X. Guan, Q. Zheng, T. Liu, C. Shen, and Z. Yang. Directed test suite augmentation via exploiting program dependency. In *Proceedings of the 6th International Workshop on Constraints in Software Testing, Verification, and Analysis*, pages 1–6, 2014.

[204] Y. Wang, L. Qiao, C. Xu, Y. Liu, S.-C. Cheung, N. Meng, H. Yu, and Z. Zhu. Hero: On the chaos when path meets modules. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 99–111. IEEE, 2021.

[205] Y. Wang, M. Wen, Y. Liu, Y. Wang, Z. Li, C. Wang, H. Yu, S.-C. Cheung, C. Xu, and Z. Zhu. Watchman: Monitoring dependency conflicts for python library ecosystem. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 125–135, 2020.

[206] Y. Wang, M. Wen, Z. Liu, R. Wu, R. Wang, B. Yang, H. Yu, Z. Zhu, and S.-C. Cheung. Do the dependency conflicts in my project matter? In *Proceedings of the 2018 26th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, pages 319–330, 2018.

[207] Y. Wang, M. Wen, R. Wu, Z. Liu, S. H. Tan, Z. Zhu, H. Yu, and S.-C. Cheung. Could i have a stack trace to examine the dependency conflict issue? In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 572–583. IEEE, 2019.

[208] Y. Wang, R. Wu, C. Wang, M. Wen, Y. Liu, S.-C. Cheung, H. Yu, C. Xu, and Z.-l. Zhu. Will dependency conflicts affect my program's semantics. *IEEE Transactions on Software Engineering*, 2021.

[209] D. Weyns and T. Holvoet. An architectural strategy for self-adapting systems. In *International Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'07)*, pages 3–3. IEEE, 2007.

[210] D. Weyns, M. U. Iftikhar, D. Hughes, and N. Matthys. Applying architecture-based adaptation to automate the management of internet-of-things. In *European Conference on Software Architecture*, pages 49–67. Springer, 2018.

[211] D. Weyns, S. Malek, and J. Andersson. Forms: Unifying reference model for formal specification of distributed self-adaptive systems. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 7(1):1–61, 2012.

[212] D. Weyns and F. Oquendo. An architectural style for self-adaptive multi-agent systems. *arXiv preprint arXiv:1909.03475*, 2019.

[213] Z. Xu, M. B. Cohen, W. Motycka, and G. Rothermel. Continuous test suite augmentation in software product lines. In *Proceedings of the 17th International Software Product Line Conference*, pages 52–61, 2013.

[214] Z. Xu, Y. Kim, M. Kim, and G. Rothermel. A hybrid directed test suite augmentation technique. In *2011 IEEE 22nd International Symposium on Software Reliability Engineering*, pages 150–159. IEEE, 2011.

[215] Z. Xu, Y. Kim, M. Kim, G. Rothermel, and M. B. Cohen. Directed test suite augmentation: techniques and tradeoffs. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 257–266, 2010.

[216] Z. Xu and G. Rothermel. Directed test suite augmentation. In *2009 16th Asia-Pacific Software Engineering Conference*, pages 406–413. IEEE, 2009.

[217] J. Xuan, B. Cornu, M. Martinez, B. Baudry, L. Seinturier, and M. Monperrus. Dynamic analysis can be improved with automatic test suite refactoring. *arXiv preprint arXiv:1506.01883*, 2015.

[218] J. Xuan, X. Xie, and M. Monperrus. Crash reproduction via test case mutation: let existing test cases help. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 910–913, 2015.

[219] H. Yan, D. Garlan, B. Schmerl, J. Aldrich, and R. Kazman. DiscoTect: A System for Discovering Architectures from Running Systems. In *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*, pages 470–479. IEEE, 2004.

[220] R. Yokomori, N. Yoshida, M. Noro, and K. Inoue. Extensions of component rank model by taking into account for clone relations. In *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, volume 3, pages 30–36. IEEE, 2016.

[221] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: a survey. *Software testing, verification and reliability*, 22(2):67–120, 2012.

[222] T. Yu. Taco: test suite augmentation for concurrent programs. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 918–921, 2015.

[223] T. Yu, Z. Huang, and C. Wang. Contesa: Directed test suite augmentation for concurrent software. *IEEE Transactions on Software Engineering*, 2018.

[224] Z. Yu, C. Bai, and K.-Y. Cai. Mutation-oriented test data augmentation for gui software fault localization. *Information and Software Technology*, 55(12):2076–2098, 2013.

[225] L. L. Zhang, C.-J. M. Liang, Y. Liu, and E. Chen. Systematically testing background services of mobile apps. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 4–15. IEEE, 2017.

[226] Y. Zheng, C. Cu, and R. N. Taylor. Maintaining architecture-implementation conformance to support architecture centrality: From single system to product line development. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 27(2):8, 2018.

[227] Y. Zheng and R. N. Taylor. Enhancing architecture-implementation conformance with change management and support for behavioral mapping. In *Proceedings of the 34th International Conference on Software Engineering*, pages 628–638. IEEE Press, 2012.