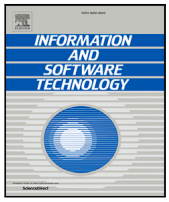
Information and Software Technology 176 (2024) 107550

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof



Bringing architecture-based adaption to the mainstream

Negar Ghorbani *, Joshua Garcia, Sam Malek

University of California, Irvine, CA, USA

ARTICLE INFO

Keywords:
Software architecture
Java modules
Adaptive framework

ABSTRACT

Software architecture has been shown to provide an appropriate level of granularity for 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 of making non-trivial changes to the manner in which a software system is implemented. In this paper, we present Acadia—a framework that automatically enables architecture-based adaptation of practically any Java 9+ application without requiring any changes to the implementation of the application itself. Acadia builds on the *Java Platform Module System (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.

1. Introduction

Development of (self-)adaptive software systems is known to be significantly more complex than that of non-adaptive systems [1]. 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 [2–6]. A software system's architecture represents the elements comprising the system (e.g., components, connectors, interfaces) and their relationships to one another [7]. *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 [8], C2 [9], Darwin [10,11], ArchJava [12], and Prism-MW [13]), 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 [12], developers would need to implement Java applications using syntax that is specific to ArchJava. For example, an ArchJava's component is defined using the "component" keyword before class declaration (`public component class nameOfClassHere`), while a component's interfaces are defined using the "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, as 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 [14]. The majority of these exemplars, however, are essentially still toy applications. Indeed, to this date, the majority of publications in this domain,

* Corresponding author.
E-mail addresses: negargh@uci.edu (N. Ghorbani), joshug4@uci.edu (J. Garcia), malek@uci.edu (S. Malek).

https://doi.org/10.1016/j.infsof.2024.107550
Received 6 July 2023; Received in revised form 15 August 2024; Accepted 16 August 2024
Available online 20 August 2024

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 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) [15], 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 paper, 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 execute various adaptation strategies with very low overhead.

Acadia offers profound and practical contributions to the software engineering research community by enabling the evaluation of the developed techniques 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. An implementation of the proposed technique is publicly available [16].

The remainder of this paper is organized as follows. Section 2 provides background on the Java module system. Section 3 describes Acadia and its implementation in detail. Section 4 demonstrates an application of Acadia in terms of an example adaptation strategy built upon it. Section 5 presents the experimental evaluation of Acadia's adaptation capabilities. The paper concludes with an outline of related research.

2. Architecture-based development in Java

In this section, we first introduce the concepts behind the Java Platform Module System (JPMS) [15], followed by the description of an illustrative example used throughout the paper for explanation of our work.

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.

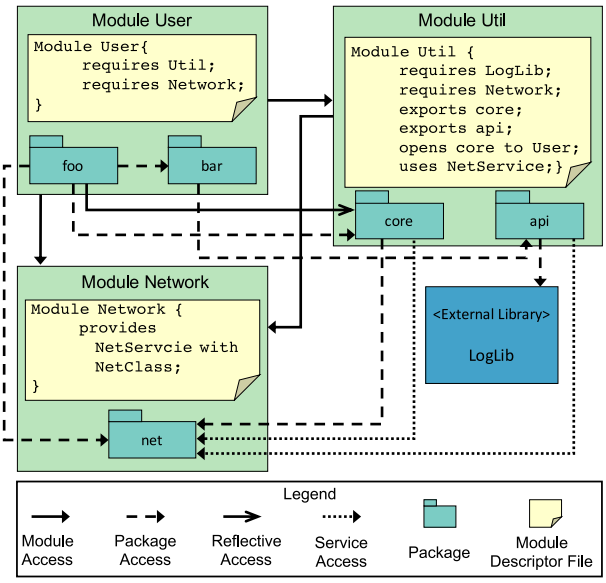


Fig. 1. An example Java app implemented using JPMS.

The main goals of modular development in JPMS include reliable configuration, strong encapsulation, scalability, platform integrity, and improved performance [17]. 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. The more refined accessibility control in JPMS 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 [18].

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 [17]. 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 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 uses [19].

2.2. Illustrative example

Fig. 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

[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 paper.

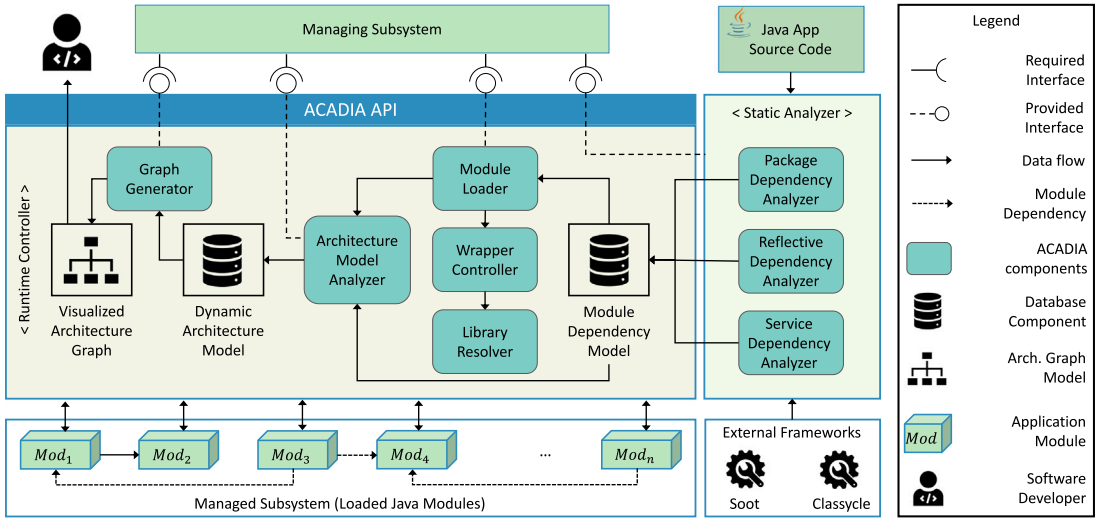


Fig. 2. A high-level overview of Acadia framework.

`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.

These directives outline an overview of the dependencies between the system's modules and their packages. Fig. 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*.

3. Acadia

Fig. 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.

3.1. Acadia's API

Acadia takes a Java 9+ application's source code as input and provides a specific API with a set of commands usable by an adaptation strategy or software developer. The adaptation strategy is denoted by *Managing Subsystem* in Fig. 2.[3] More specifically, Acadia can execute various architecture-based adaptation operations on the running Java system, denoted by *Managed Subsystem* in Fig. 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 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 3.3.

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 its implementation. Acadia relies on an existing static analysis approach, represented as *Package Dependency Analyzer* in Fig. 2, to identify the dependencies between all packages of a Java application. We leveraged Classycle [22] 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 [23], represented as *Reflective Dependency Analyzer* in Fig. 2. It analyzes the source code of the Java app and extracts all instances of reflective dependencies [24]. Fig. 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.

Java applications can contain implementations of service providers using `ServiceLoader` API, which indicates the dependencies of types *provides* and *uses* directives. Fig. 4 shows an example of a service

[3] We have followed the terminology used in several prior (self-)adaptation publications (e.g., [20,21]) that view an adaptive system to be consisting of two subsystems: a *managed subsystem* that provides the core features and functionalities for solving the domain problems, and a *managing subsystem* that observes the managed subsystem and effects adaptation decisions to satisfy certain goals.

Table 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.

```
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 }
```

Fig. 3. An example of a reflective dependency in Java.

dependency in Java [25] obtained from Java documentation [26]. 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 8–10). To identify such dependencies, we developed another custom static analysis, using the Soot framework [23], shown as *Service Dependency Analyzer* in Fig. 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 [24].

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. More specifically, the *Module Dependency Model* database stores the class-level and package-level dependencies of the managed subsystem in terms of mappings between them. In addition, the *Module Dependency*

```
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      public static Encoder getEncoder(String
           encodingName) {
8          for (CodecSet cp : codecSetLoader) {
9              Encoder enc = cp.getEncoder(encodingName);
10             if (enc != null) {...}
11         }
12     }
13 }
```

Fig. 4. An example of service dependency in Java.

```
1  import java.lang.*
2  import java.lang.module.*
3
4  protected static ModuleLayer.Controller
       createModuleWrapper(String moduleName, String
       modulePath Path[] libPathsArray){
5      ModuleFinder finder = ModuleFinder.of(Paths.get(
           modulePath));
6      ModuleLayer parent = Launcher.class.getModule().
           getLayer();
7      ModuleLayer.Controller moduleWrapper = null;
8
9      ModuleFinder libraryFinder = ModuleFinder.of(
           libPathsArray);
10     Configuration cf = parent.configuration().resolve(
           finder, libraryFinder, Set.of(moduleName));
11     ClassLoader scl = ClassLoader.getSystemClassLoader
           ();
12     moduleWrapper = ModuleLayer.
           defineModulesWithOneLoader(cf, List.of(parent)
           , scl);
13
14     return moduleWrapper;
15 }
```

Fig. 5. A simplified code snippet including the implementation of a wrapper using java.lang.ModuleLayer in JPMS.

Model database can change. If 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 1) to invoke the *Static Analyzer*, extract that information, and update the *Module Dependency Model*.

3.3. Runtime Controller

The *Runtime Controller* compartment is responsible for the implementation of Acadia's runtime API, listed in Table 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 [27], which defines a group of modules in the Java virtual machine (JVM). Fig. 5 shows a simplified code snippet of a module wrapper's implementation that uses JPMS built-in classes. More specifically, to create a wrapper, a `ModuleLayer` is initiated followed by loading the target module and its required libraries into the `ModuleLayer`.

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 [28]. 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 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 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. Fig. 6 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. Note that *Module Dependency Model* is a database that stores static dependencies of the managed system. In contrast, the *Dynamic Architecture Model* is responsible for capturing the dynamic architecture of the managed system. In other words, the *Module Dependency Model* contains a superset of all dependencies, as opposed to an architectural description model.

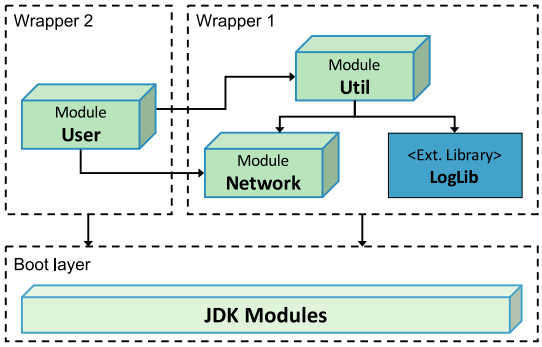


Fig. 6. The managed subsystem after loading all the example app's modules (Section 2.2).

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 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 [29]. 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 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.

Fig. 7 demonstrates the multidimensional network representation of the architectural model of our illustrative example application, explained in Section 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.

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. 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 [29]. In other words, the dependent nodes *(u)* on a specific node v in a specific set of dimensions can be formally specified, as shown in Eq. (1).

$$Dependent\_Nodes(v, D) \equiv \{v \in V \mid \exists (u, v, d) \in E \wedge d \in D\} \tag{1}$$

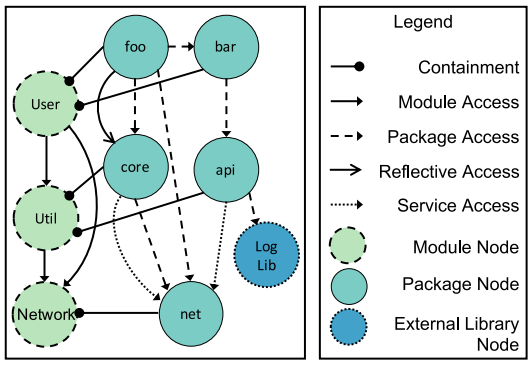


Fig. 7. The multidimensional network representing the dynamic architecture model of the illustrative example in Section 2.2 after loading all modules.

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 [30], i.e., a widely-used graph description language. Furthermore, in the case of the `visualize` command, it uses visualization libraries, such as Graphviz [31], to visualize the generated graph in DOT language for human perception.

4. Application of ACADIA

Kramer and Magee have defined an influential reference architecture model for self-adaptive systems [32] consisting of three layers: *Component Control*, *Change Management*, and *Goal Management*. The bottom layer is Component Control, which provides the operations for making changes to the managed software. The middle layer is Change Management, which provides sequencing to ensure changes are applied in a manner that does not jeopardize the integrity of the system. The top layer is Goal Management, which provides the adaptation logic of determining what changes, if any, should be made to the managed system to satisfy certain objectives.

From the perspective of this three-layer reference architecture, ACADIA's contribution is in the Component Control layer in that it provides support for effecting changes to a Java 9+ software system without requiring any changes to its implementation. ACADIA can be used in the construction of different (self-)adaptive systems in Java. In fact, ACADIA is completely independent of the Change Management and Goal Management capabilities that are implemented on top of it. However, to showcase and evaluate ACADIA, we needed actual functioning (self-)adaptive software systems to validate our claims that (1) ACADIA can be used to adapt Java 9+ software systems without requiring changes to their implementation, and (2) measure the overhead imposed by the use ACADIA in such systems. To that end, we used ACADIA to develop an exemplary self-adaptation logic aimed at keeping the memory usage of Java 9+ applications below a user-provided threshold. Memory efficiency is particularly critical for the 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.[4] 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 1. It uses ACADIA's API commands to execute its adaptation operations. Note that the adaptation strategy explained in Algorithm 1 is not a built-in strategy and is solely defined as an example to showcase ACADIA's adaptation capabilities.

Algorithm 1: 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 ← getModules(app.thread.getStackTrace())
4          foreach module ∈ App.loadedModules() do
5              if module ∉ RunningModules and
                 ACADIA.getDependent(module) ∉ RunningModules then
6                  ACADIA.unload(module)
7              end
8              if module.isLoaded() and hasReplacement(module) then
9                  NewModule ← module.getReplacement()
10                 DependentModules ← 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 ← getModules(app.thread.getStackTrace())
21     if App.loadedModules ≠ RequiredModules then
22         ACADIA.load(RequiredModules - App.loadedModules())
23     end
24 end
```

Algorithm 1 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 1 first retrieves the list of running methods using the stack trace of the app's running thread, and consequently, obtains the list of modules being used and loaded 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 1 checks whether the module can be offloaded safely. In other words, it checks if the target module is not currently running. Algorithm 1 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 1 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

[4] 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.

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 1 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 each module is considered 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). If any of these required modules are not loaded (line 21), the algorithm uses Acadia's `load` command to load them (line 22) and eventually terminates when the application thread stops.

The adaptation strategy described here is only applicable to stateless modules. It does not have a mechanism to guarantee the application does not fall into an inconsistent state when modules are stateful. Our adaptation strategy can be extended to support stateful modules by implementing one of the existing solutions described in the literature, such as Quiescence [33], Tranquility [34], or even our own work called Savasana [35]. Such techniques can be used to ensure adaptations do not result in application inconsistencies when the modules are stateful.

5. Evaluation

Our evaluation considers 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 execution time overhead?

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 [36]. Using these applications, we designed an experiment in which we execute the adaptation strategy explained in Section 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.

5.1. Subject applications

To examine the objectives of Acadia, we used three different open-source Java applications on GitHub [36], namely Quasar [37], Tascalate JavaFlow [38], and Rhizomatic [39]. 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 [40] provides asynchronous programming tools for Java and Kotlin and consists of 4 modules and 33 packages, implemented with 50.6K lines of Java code. It also depends on three external libraries. Fig. 8 shows a high-level overview of Quasar's architecture, including its modules and required libraries.

The second application, Tascalate JavaFlow [38], contains libraries that provide Java developers with representation of the program's control state. Tascalate JavaFlow consists of 10 modules, containing 27 packages in total, implemented with 11.9K lines of Java code. Fig. 9 demonstrates the module-level architecture of Tascalate JavaFlow.

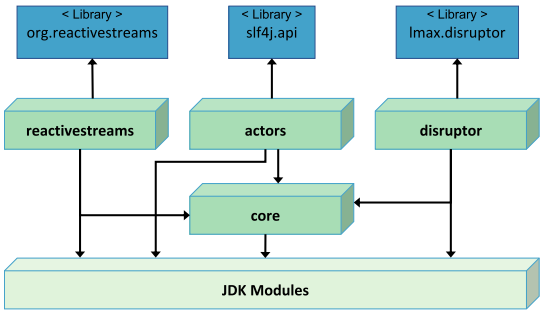


Fig. 8. Quasar's architecture graph in module-level.

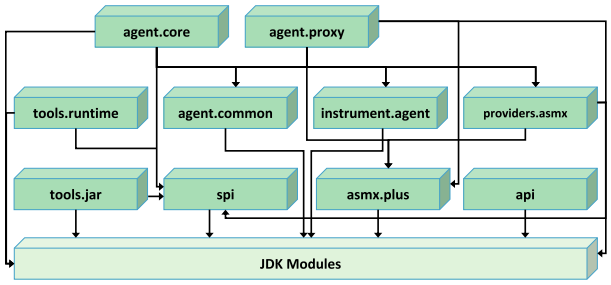


Fig. 9. Tascalate's architecture graph in module-level.

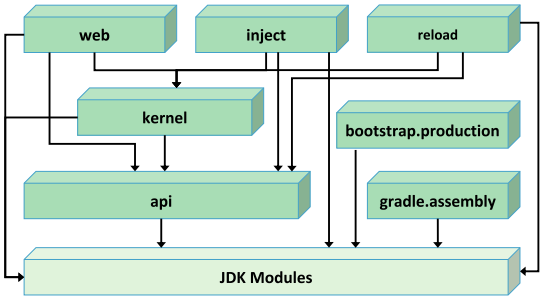


Fig. 10. Rhizomatic's architecture graph in module-level.

The third application, Rhizomatic [39], is a runtime environment built on JPMS that provides different programming models for developing web and RESTful Java apps. Rhizomatic consists of 7 modules and 157 packages, implemented with 3.9K lines of Java code. Fig. 10 illustrates a high-level overview of Rhizomatic's module architecture.

5.2. RQ1: Adaptation effectiveness

To assess Acadia's effectiveness, we applied the adaptation strategy described in Section 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 [41], 12 example scenarios implemented for Tascalate [42], and 5 example scenarios for Rhizomatic [43]. 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

Table 2
Results of Acadia performing an adaptation strategy on different example scenarios of three subject Java apps.

App name	Example scenario	# Mod.	AVG. Mem. (MB) Before	After	Δ %	Adapt. Succ.
	Fiber	1	51	24.4	52.16	✓
	FiberAsync	1	55	28.4	48.36	✓
	IO	1	59	32.4	45.08	✓
	Channel	1	52	24.8	52.31	✓
Quasar	Actor	2	56	36.8	34.29	✓
	SrvrBhvr	3	60	41.8	30.33	✗
(Max Mem.:	ProxySrvr	2	52	33.8	35.00	✓
62 MB	EventSrc	2	51	29.6	41.96	✓
Mem. Thold:	Supervisor	2	58	35	39.66	✓
40 MB)	ReactStrms	2	62	40.6	34.52	✗
	Issues	2	59	39.8	32.54	✓
	Migration	2	52	33.4	35.77	✓
	PingPong	2	54	34.8	35.56	✓
	GenEvent	3	60	40.8	32.00	✗
	GenServer	2	54	32.2	40.37	✓
	GalaxySrvr	2	52	29.6	43.08	✓
Average runtime memory size reduction:						39.56%
Adaptation success rate:						81%
	cdi-owb	1	11	3.2	70.91	✓
	cdi-weld3	1	12	3.2	73.33	✓
Tascalate	cd-weld4	1	12	3.2	73.33	✓
	cglib	2	12	5.4	55.00	✓
(Max Mem.:	common	2	12	5.8	51.67	✓
12 MB	dyninvoke	3	12	6.42	46.43	✗
Mem. Thold:	jee	1	12	3.2	73.33	✓
6 MB)	lambdas	1	11	3.2	70.91	✓
	retro-lmbds	1	12	3.2	73.33	✓
	skynet	1	12	3.2	73.33	✓
	trampoline6	3	12	6.42	46.43	✗
	trampoline9	3	12	7.4	38.33	✗
Average runtime memory size reduction:						62.20%
Adaptation success rate:						75%
Rhizomatic	BTS-msg	1	47	19.6	58.30	✓
(Max Mem.:	BTS-msg-gdl	1	47	19.6	58.30	✓
47 MB	BTS-msg-prd	1	43	18.6	56.74	✓
Mem. Thold:	msg-api	1	47	18.6	60.43	✓
20 MB)	msg-srvc	3	46	24.42	46.89	✗
Average runtime memory size reduction:						56.13%
Adaptation success rate:						80%

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 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 clear the occupied memory. We also changed the certain modules' implementations 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 applications' states 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 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 specified the number of affected modules (*# Mod.* column), and measured the average runtime memory size in Megabytes (MB) throughout the execution of a scenario. For each example scenario, Table 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 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 reload 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.

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 3. Table 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 s 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 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 4), i.e., about 5–8% on average.

5.4. RQ3: Overhead on applications' execution

As described in Section 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`

Table 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

Table 4
Results for the overhead on the subject apps' execution time.

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%

that implements a module wrapper) and when the modules are loaded into separate module wrappers (i.e., with the overhead of wrappers). Table 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 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 [44].

Overall, our experiments indicate that Acadia can effectively enable architecture-based adaptation in Java 9+ applications without requiring any change to their implementation, and with a low overhead in terms of execution time.

Since the subject applications in our evaluation were not adaptive on their own, we could not compare the overhead posed for adaptation using Acadia against a setup where the same adaptations were performed in these applications without Acadia.

6. Threats to validity

The main threat to internal validity is that static analysis approaches used in the Acadia's implementation have the inherent risk of false positives. Since Acadia takes the results of Classycle and Soot as input for the Module Dependency Model, it inherits all of their limitations. False positives or negatives in the results of the static analysis tools may cause Acadia to build an inaccurate module dependency and dynamic architecture models of the application, which may lead to failure of the managing subsystem's adaptation strategy. However, the static analysis frameworks we used are among the widely used tools in the literature. Classycle has been used and in development for over 11 years and leveraged by other state-of-the-art tools for software architecture [45–51]. Soot is a widely used [52,53] and actively maintained framework [54] for static analysis of Java programs.

The main threat to external validity is the selection and number of Java applications in our evaluation dataset. To mitigate this threat, we selected relatively large open-source Java 9+ applications from GitHub, one of the largest and most widely used open-source repositories online. Another external threat to validity is that Acadia is specific to Java 9+ and does not apply to other languages, as it is implemented on top of JPMS and its run-time constructs. This threat is alleviated by the fact that Java is one of the most widely used languages in the world [55,56]. However, the general idea behind Acadia can be applied to any other language with modular programming constructs. Additionally, Acadia only targets one type of adaptation strategy, i.e., architecture-based adaptation.

7. Related work

This section overviews prior work on dynamic software adaptation and architecture-based software adaptation. At the outset, we note that there are many facets to software adaptation. Several prior studies (e.g., [57–59]), including those by one of the authors (e.g., [60,61]), have explored the variability points in dynamic adaptation. Prior

studies have also enumerated the challenges of engineering adaptive software [1,20]. The focus of this work is on the implementation aspects of adaptive software. To evaluate our research, we have applied Acadia to a particular adaptation problem, but this is merely to evaluate our claims that (1) Acadia can be used to adapt existing real-world Java software systems without making any changes to their implementation, and (2) measure the overhead associated with Acadia. There are numerous other key concerns in (self-)adaptation, including for example how the adaptation is triggered, how the adaptation decisions are made, and whether certain assurances can be provided about the adaptation. While we acknowledge these concerns, they are not the focus of this work. We point the interested reader to the above-mentioned publications for an in-depth explanation of factors that should be considered in adaptation.

Since our focus is on implementation techniques, in the following sections we first discuss implementation techniques broadly, including those that are targeting service-oriented, cloud, or product line environments. We then provide an overview of architecture-based adaptation implementation techniques, as they are more relevant to Acadia's architecture-based approach.

Dynamic Adaptation Implementation: Dynamic adaptability has become one of the major properties of large software applications in many different domains [62]. The software engineering community has proposed various techniques for dynamic adaptation of software systems. Some address dynamic adaptation by providing models to specify and tools to implement dynamically adaptive software systems [63–67]: They introduce models to specify and execute adaptive systems [64–66] or propose a methodology for modeling and validation of dynamic adaptation [67]; and studies that focus on dynamic adaptation in service-based applications [68] by providing a runtime support [69] or a framework to specify and analyze dynamically configurable service architectures [70]. iPOJO [71] proposes a service-oriented component framework for handling dynamic adaptation. Klus et al. [72] 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 [73,74], investigating safe adaptation [75], utilizing dynamic product line architecture [76–78], using software process technology [79], and addressing dynamic adaptation from a security perspective [80].

Architecture-Based Adaptation Implementation: 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 [2,81], e.g., presenting an architectural strategy for self-adapting systems [82] or a reference architecture for system configuration and behavior adaptation [83].

Various approaches have leveraged architectural models to address dynamic adaptation of software systems [3,84–86]. Particularly, Cheng et al. [3] utilize externalized architectural models for adaptation of pervasive computing systems based on performance-related criteria. Bencomo et al. [84] use architectural models for representation, generation, and operation of highly configurable component-based systems. Additionally, Cu et al. [85] 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. [86] 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 [8], C2 [9], ArchJava [12], Darwin [10,11], Prism-MW [13], and ActivFORMS [87]. For instance, Rainbow [8] uses architecture models and styles to provide a reusable infrastructure that monitors a running system and updates its architectural model based on different self-adaptation strategies. More recently, Swanson et al. developed Refract, which extends Rainbow with new components and algorithms targeting failure avoidance [88]. The nature of adaptations provided by our approach is not fundamentally different from these seminal works, many of which support adaptation of components and their interfaces (sometimes referred to as ports). Some of the approaches, such as C2, support adaptation of connectors, which our approach does not, because the notion of connector is still not explicit in the JPMS.

Prior studies have investigated architecture-based adaptation in other contexts, e.g., service-based applications [89–92], distributed systems [93,94], microservices [95], internet-of-things (IoT) [96], and mobile computing [97]. More specifically, Weyns et al. [93] present an architectural style for decentralized self-adaptive systems. Similar to Acadia, Florio et al. [95] introduce Gru that adds adaptation capabilities to microservices without changing their implementations. In the context of IoT, Weyns et al. [96] propose an architecture-based adaptation approach for automating IoT systems' management. Hallsteinsen et al. [97] introduce MADAM, a generic middleware for mobile apps' adaptation. Other approaches in container-based applications require migration to specific platforms, e.g., OSGi [98], or target different goals, e.g., easier collaboration [99] or freezing of an application's architecture [90].

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).

8. Conclusion

The capability to retain an accurate representation of a 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 software development by requiring non-trivial modification to the manner it is developed. This paper introduces Acadia, a framework that leverages JPMS and static analysis techniques to automatically enable architecture-based adaptation support in any Java 9+ application. More specifically, Acadia provides support for making changes to Java 9+ software systems, i.e., Component Control layer [32], without requiring any changes to their implementation and can be used in the construction of different (self-)adaptive systems, as it is completely independent of the managing systems that are implemented on top of it, i.e., Change and Goal Management layers [32]. 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.

An interesting avenue of future work will be to study the extent to which ideas underlying Acadia can be applied in other environments. For instance, C++ has also recently introduced support for modules in its 20th edition [100]. In our future work, we aim to investigate the degree to which a framework similar to Acadia can be developed for C++.

CRediT authorship contribution statement

Negar Ghorbani: Writing – review & editing, Writing – original draft, Visualization, Validation, Software, Methodology, Formal analysis, Data curation, Conceptualization. **Joshua Garcia:** Writing – review & editing, Supervision, Funding acquisition. **Sam Malek:** Writing – review & editing, Supervision, Funding acquisition, Conceptualization.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Negar Ghorbani reports financial support was provided by National Science Foundation. Co-authors, Joshua Garcia and Sam Malek have conflicts of interest with Yuanfang Cai on the editorial board due to previous collaborations. The corresponding author, Negar Ghorbani, was employed at Meta Platforms Inc. after finishing the manuscript, but before the submission.

Data availability

The data and the source code of the tool have been shared via a website mentioned in the paper's references.

Acknowledgments

This work was supported in part by awards CNS-1823262, CCF-2106306, CCF-2211790 from the US National Science Foundation.

References

[1] 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, J. Whittle, Software engineering for self-adaptive systems: A research roadmap, in: B.H.C. Cheng, R. de Lemos, H. Giese, P. Inverardi, J. Magee (Eds.), Software Engineering for Self-Adaptive Systems, Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, pp. 1–26, [Online]. Available: https://doi.org/10.1007/978-3-642-02161-9_1.

[2] P. Oreizy, R.N. Taylor, On the role of software architectures in runtime system reconfiguration, IEE Proc.-Softw. 145 (5) (1998) 137–145.

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

[4] N. Subramanian, L. Chung, Software architecture adaptability: an NFR approach, in: Proceedings of the 4th International Workshop on Principles of Software Evolution, 2001, pp. 52–61.

[5] P. Oreizy, N. Medvidovic, R.N. Taylor, Runtime software adaptation: framework, approaches, and styles, in: Companion of the 30th International Conference on Software Engineering, 2008, pp. 899–910.

[6] P. Tarvainen, Adaptability evaluation at software architecture level, Open Softw. Eng. J. 2 (1) (2008).

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

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

[9] E.M. Dashofy, A. Van der Hoek, R.N. Taylor, Towards architecture-based self-healing systems, in: Proceedings of the First Workshop on Self-Healing Systems, 2002, pp. 21–26.

[10] J. Magee, N. Dulay, S. Eisenbach, J. Kramer, Specifying distributed software architectures, in: Software Engineering—ESEC'95: 5th European Software Engineering Conference Sitges, Spain, September 25–28, 1995 Proceedings 5, Springer, 1995, pp. 137–153.

[11] J. Magee, J. Kramer, Dynamic structure in software architectures, ACM SIGSOFT Softw. Eng. Notes 21 (6) (1996) 3–14.

[12] J. Aldrich, C. Chambers, D. Notkin, ArchJava: Connecting software architecture to implementation, in: Proceedings of the 24th International Conference on Software Engineering, ICSE '02, Association for Computing Machinery, New York, NY, USA, 2002, pp. 187–197, [Online]. Available: https://doi.org/10.1145/581339.581365.

[13] S. Malek, M. Mikic-Rakic, N. Medvidovic, A style-aware architectural middleware for resource-constrained, distributed systems, IEEE Trans. Softw. Eng. 31 (3) (2005) 256–272.

[14] Exemplars - Self-adaptive systems artifacts and model problems, 2021, https://www.hpi.uni-potsdam.de/giese/public/selfadapt/exemplars/. (Accessed 20 January 2022).

[15] Project jigsaw, 2017, https://openjdk.java.net/projects/jigsaw/. (Accessed 10 January 2022).

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

[17] Understanding java 9 modules, 2017, https://www.oracle.com/corporate/features/understanding-java-9-modules.html. (Accessed 10 January 2022).

[18] The java platform module system (JSR 376), 2017, https://cr.openjdk.java.net/~mr/jigsaw/spec/. (Accessed 10 January 2022).

[19] J. Gosling, B. Joy, G. Steele, G. Bracha, A. Buckley, D. Smith, The java language specification, java SE 9 edition, 2017, https://docs.oracle.com/javase/specs/jls/se9/html/index.html. (Accessed 10 January 2022).

[20] R. de Lemos, H. Giese, H.A. Müller, M. Shaw, J. Andersson, M. Litoiu, B. Schmerl, G. Tamura, N.M. Villegas, T. Vogel, D. Weyns, L. Baresi, B. Becker, N. Bencomo, Y. Brun, B. Cukic, R. Desmarais, S. Dustdar, G. Engels, K. Geihs, K.M. Göschka, A. Gorla, V. Grassi, P. Inverardi, G. Karsai, J. Kramer, A. Lopes, J. Magee, S. Malek, S. Mankovskii, R. Mirandola, J. Mylopoulos, O. Nierstrasz, M. Pezzè, C. Prehofer, W. Schäfer, R. Schlichting, D.B. Smith, J.P. Sousa, L. Tahvildari, K. Wong, J. Wuttke, Software engineering for self-adaptive systems: A second research roadmap, in: R. de Lemos, H. Giese, H.A. Müller, M. Shaw (Eds.), Software Engineering for Self-Adaptive Systems II: International Seminar, Dagstuhl Castle, Germany, October 24-29, 2010 Revised Selected and Invited Papers, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 1–32, [Online]. Available: https://doi.org/10.1007/978-3-642-35813-5_1.

[21] N. Abbas, J. Andersson, D. Weyns, ASPLe: A methodology to develop self-adaptive software systems with systematic reuse, J. Syst. Softw. 167 (2020) 110626, [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121220301047.

[22] F.-J. Elmer, Classycle: Analysing tools for java class and package dependencies, in: How Classycle Works, 2012.

[23] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, V. Sundaresan, Soot: A java bytecode optimization framework, in: CASCON First Decade High Impact Papers, IBM Corp., 2010, pp. 214–224.

[24] N. Ghorbani, J. Garcia, S. Malek, Detection and repair of architectural inconsistencies in java, in: 2019 IEEE/ACM 41st International Conference on Software Engineering, ICSE, IEEE, 2019, pp. 560–571.

[25] ServiceLoader (java platform SE 9 & JDK 9), 2022, https://docs.oracle.com/javase/9/docs/api/java/util/ServiceLoader.html.

[26] ServiceLoader (java platform SE 8), 2024, https://docs.oracle.com/javase/8/docs/api/java/util/ServiceLoader.html.

[27] ModuleLayer (java SE 15 & JDK 15), 2020, https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/lang/ModuleLayer.html. (Accessed 10 January 2022).

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

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

[30] DOT language, 2021, https://graphviz.org/doc/info/lang.html. (Accessed 10 January 2022).

[31] E.R. Gansner, S.C. North, An open graph visualization system and its applications to software engineering, Softw. - Pract. Exp. 30 (11) (2000) 1203–1233.

[32] J. Kramer, J. Magee, Self-managed systems: an architectural challenge, in: Future of Software Engineering, FOSE'07, IEEE, 2007, pp. 259–268.

[33] J. Kramer, J. Magee, The evolving philosophers problem: dynamic change management, IEEE Trans. Softw. Eng. 16 (11) (1990) 1293–1306.

[34] Y. Vandewoude, P. Ebraert, Y. Berbers, T. D'Hondt, Tranquility: A low disruptive alternative to quiescence for ensuring safe dynamic updates, IEEE Trans. Softw. Eng. 33 (12) (2007) 856–868.

[35] A. Sadeghi, N. Esfahani, S. Malek, Ensuring the consistency of adaptation through inter- and intra-component dependency analysis, ACM Trans. Softw. Eng. Methodol. 26 (1) (2017) [Online]. Available: https://doi.org/10.1145/3063385.

[36] GitHub, 2022, https://github.com.

[37] Puniverse/quasar, 2013, https://github.com/puniverse/quasar. (Accessed 10 January 2022).

[38] Tascalate JavaFlow, 2017, https://github.com/vsilaev/tascalate-javaflow. (Accessed 10 January 2022).

[39] Rhizomatic, 2018, https://github.com/Rhizomatic-IO/rhizomatic. (Accessed 10 January 2022).

[40] Quasar, 2018, http://docs.paralleluniverse.co/quasar/. (Accessed 10 January 2022).

[41] Quasar examples, 2018, http://docs.paralleluniverse.co/quasar/#examples. (Accessed 10 January 2022).

[42] Tascalate JavaFlow examples, 2018, https://github.com/vsilaev/tascalate-javaflow-examples. (Accessed 10 January 2022).

[43] Rhizomatic samples, 2018, https://github.com/Rhizomatic-IO/rhizomatic-samples. (Accessed 10 January 2022).

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

[45] M.R. Shaheen, L. du Bousquet, Quantitative analysis of testability antipatterns on open source java applications, in: QAOOSE 2008-Proceedings, 2008, p. 21.

[46] R. Yokomori, N. Yoshida, M. Noro, 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, Vol. 3, IEEE, 2016, pp. 30–36.

[47] E. Constantinou, G. Kakarontzas, I. Stamelos, Towards open source software system architecture recovery using design metrics, in: 2011 15th Panhellenic Conference on Informatics, 2011, pp. 166–170.

[48] D.M. Le, P. Behnamghader, J. Garcia, D. Link, A. Shahbazian, 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, IEEE, 2015, pp. 235–245.

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

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

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

[52] L. Hendren, Uses of the soot framework, 2024, http://www.sable.mcgill.ca/~hendren/sootusers/.

[53] P. Lam, E. Bodden, O. Lhoták, L. Hendren, The soot framework for java program analysis: a retrospective, in: Cetus Users and Compiler Infastructure Workshop, CETUS 2011, Vol. 15, 2011, p. 35.

[54] Soot GitHub issue, 2024, https://github.com/Sable/soot/issues.

[55] TIOBE index for May 2024, 2024, https://www.tiobe.com/tiobe-index/.

[56] The state of the octoverse 2024, 2024, https://octoverse.github.com/.

[57] M. Salehie, L. Tahvildari, Self-adaptive software: Landscape and research challenges, ACM Trans. Auton. Adapt. Syst. 4 (2) (2009) [Online]. Available: https://doi.org/10.1145/1516533.1516538.

[58] F.D. Macías-Escrivá, R. Haber, R. del Toro, V. Hernandez, Self-adaptive systems: A survey of current approaches, research challenges and applications, Expert Syst. Appl. 40 (18) (2013) 7267–7279, [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0957417413005125.

[59] C. Krupitzer, F.M. Roth, S. VanSyckel, G. Schiele, C. Becker, A survey on engineering approaches for self-adaptive systems, Pervasive Mob. Comput. 17 (2015) 184–206, [Online]. Available: https://www.sciencedirect.com/science/article/pii/S157411921400162X, 10 years of Pervasive Computing' In Honor of Chatschik Bisdikian.

[60] J. Andersson, R. de Lemos, S. Malek, D. Weyns, Modeling dimensions of self-adaptive software systems, in: B.H.C. Cheng, R. de Lemos, H. Giese, P. Inverardi, J. Magee (Eds.), Software Engineering for Self-Adaptive Systems, Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, pp. 27–47, [Online]. Available: https://doi.org/10.1007/978-3-642-02161-9_2.

[61] E. Yuan, N. Esfahani, S. Malek, A systematic survey of self-protecting software systems, ACM Trans. Auton. Adapt. Syst. 8 (4) (2014) [Online]. Available: https://doi.org/10.1145/2555611.

[62] J. Fox, S. Clarke, Exploring approaches to dynamic adaptation, in: Proceedings of the 3rd International DiscCoTec Workshop on Middleware-Application Interaction, 2009, pp. 19–24.

[63] J. Andersson, R.d. Lemos, S. Malek, D. Weyns, Modeling dimensions of self-adaptive software systems, in: Software Engineering for Self-Adaptive Systems, Springer, 2009, pp. 27–47.

[64] T.E. Bihari, K. Schwan, Dynamic adaptation of real-time software, ACM Trans. Comput. Syst. (TOCS) 9 (2) (1991) 143–174.

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

[66] M. Amoui, M. Derakhshanmanesh, J. Ebert, L. Tahvildari, Achieving dynamic adaptation via management and interpretation of runtime models, J. Syst. Softw. 85 (12) (2012) 2720–2737.

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

[68] E. Di Nitto, C. Ghezzi, A. Metzger, M. Papazoglou, K. Pohl, A journey to highly dynamic, self-adaptive service-based applications, Autom. Softw. Eng. 15 (3) (2008) 313–341.

[69] A. Mukhija, D.S. Rosenblum, H. Foster, S. Uchitel, Runtime support for dynamic and adaptive service composition, in: Rigorous Software Engineering for Service-Oriented Systems, Springer, 2011, pp. 585–603.

[70] H. Foster, A. Mukhija, D.S. Rosenblum, S. Uchitel, Specification and analysis of dynamically-reconfigurable service architectures, in: Rigorous Software Engineering for Service-Oriented Systems, Springer, 2011, pp. 428–446.

[71] C. Escoffier, R.S. Hall, Dynamically adaptable applications with iPOJO service components, in: International Conference on Software Composition, Springer, 2007, pp. 113–128.

[72] H. Klus, D. Niebuhr, 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, 2007, pp. 21–28.

[73] L. Cianciaruso, F. Di Forenza, E. Di Nitto, M. Miglierina, N. Ferry, 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, IEEE, 2014, pp. 401–408.

[74] D. Weyns, S. Malek, J. Andersson, Forms: Unifying reference model for formal specification of distributed self-adaptive systems, ACM Trans. Auton. Adapt. Syst. (TAAS) 7 (1) (2012) 1–61.

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

[76] A. Helleboogh, D. Weyns, K. Schmid, T. Holvoet, K. Schelfthout, 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, Carnegie Mellon University; Pittsburgh, PA, USA, 2009, pp. 18–27.

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

[78] H. Gomaa, K. Hashimoto, Dynamic software adaptation for service-oriented product lines, in: Proceedings of the 15th International Software Product Line Conference, Volume 2, 2011, pp. 1–8.

[79] G. Valetto, G. Kaiser, G.S. Kc, A mobile agent approach to process-based dynamic adaptation of complex software systems, in: European Workshop on Software Process Technology, Springer, 2001, pp. 102–116.

[80] M. Amoud, O. Roudies, Dynamic adaptation and reconfiguration of security in mobile devices, in: 2017 International Conference on Cyber Incident Response, Coordination, Containment & Control, Cyber Incident, IEEE, 2017, pp. 1–6.

[81] R.N. Taylor, N. Medvidovic, P. Oreizy, Architectural styles for runtime software adaptation, in: 2009 Joint Working IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture, IEEE, 2009, pp. 171–180.

[82] D. Weyns, T. Holvoet, An architectural strategy for self-adapting systems, in: International Workshop on Software Engineering for Adaptive and Self-Managing Systems, SEAMS'07, IEEE, 2007, p. 3.

[83] V. Braberman, N. D'Ippolito, J. Kramer, D. Sykes, 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, 2015, pp. 9–16.

[84] N. Bencomo, P. Grace, C. Flores, D. Hughes, G. Blair, Genie, in: 2008 ACM/IEEE 30th International Conference on Software Engineering, IEEE, 2008, pp. 811–814.

[85] C. Cu, R. Culver, Y. Zheng, Dynamic architecture-implementation mapping for architecture-based runtime software adaptation, in: SEKE, 2020, pp. 135–140.

[86] N. D'Ippolito, V. Braberman, J. Kramer, J. Magee, D. Sykes, 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, 2014, pp. 688–699.

[87] M.U. Iftikhar, D. Weyns, ActivFORMS: A runtime environment for architecture-based adaptation with guarantees, in: 2017 IEEE International Conference on Software Architecture Workshops, ICSAW, IEEE, 2017, pp. 278–281.

[88] J. Swanson, M.B. Cohen, M.B. Dwyer, B.J. Garvin, 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, 2014, pp. 377–388.

[89] L. Baresi, E. Di Nitto, C. Ghezzi, S. Guinea, A framework for the deployment of adaptable web service compositions, Serv. Orient. Comput. Appl. 1 (1) (2007) 75–91.

[90] L. Baresi, S. Guinea, Self-supervising bpel processes, IEEE Trans. Softw. Eng. 37 (2) (2010) 247–263.

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

[92] A. Charfi, T. Dinkelaker, M. Mezini, A plug-in architecture for self-adaptive web service compositions, in: 2009 IEEE International Conference on Web Services, IEEE, 2009, pp. 35–42.

[93] D. Weyns, F. Oquendo, An architectural style for self-adaptive multi-agent systems, 2019, arXiv preprint arXiv:1909.03475.

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

[95] L. Florio, E. Di Nitto, Gru: An approach to introduce decentralized autonomic behavior in microservices architectures, in: 2016 IEEE International Conference on Autonomic Computing, ICAC, IEEE, 2016, pp. 357–362.

[96] D. Weyns, M.U. Iftikhar, D. Hughes, N. Matthys, Applying architecture-based adaptation to automate the management of internet-of-things, in: European Conference on Software Architecture, Springer, 2018, pp. 49–67.

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

[98] J.C. Américo, W. Rudametkin, D. Donsez, Managing the dynamism of the OSGi service platform in real-time java applications, in: Proceedings of the 27th Annual ACM Symposium on Applied Computing, 2012, pp. 1115–1122.

[99] D. Schall, C. Dorn, S. Dustdar, I. Dadduzio, Viecar-enabling self-adaptive collaboration services, in: 2008 34th Euromicro Conference Software Engineering and Advanced Applications, IEEE, 2008, pp. 285–292.

[100] Microsoft, Overview of modules in C++, 2024, https://learn.microsoft.com/en-us/cpp/cpp/modules-cpp?view=msvc-170.