Journal of Systems Architecture 160 (2025) 103343

Contents lists available at ScienceDirect

Journal of Systems Architecture

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

Component-based architectural regression test selection for modularized software systems

Mohammed Al-Refai [a,*], Mahmoud M. Hammad [b]

[a] Computer Science, Computer and Information Technology, Jordan university of science and technology, P.O. Box 3030, Irbid, 22110, Jordan
[b] Software Engineering, Computer and Information Technology, Jordan university of science and technology, P.O. Box 3030, Irbid, 22110, Jordan

ARTICLE INFO

Keywords:
Regression test selection
Static analysis
Component-based architecture
Java platform module system
Software architecture

ABSTRACT

Regression testing is an essential part of software development, but it can be costly and require significant computational resources. Regression Test Selection (RTS) improves regression testing efficiency by only re-executing the tests that have been affected by code changes. Recently, dynamic and static RTS techniques for Java projects showed that selecting tests at a coarser granularity, class-level, is more effective than selecting tests at a finer granularity, method- or statement-level. However, prior techniques are mainly considering Java object-oriented projects but not modularized Java projects. Given the explicit support of architectural constructs introduced by the *Java Platform Module System (JPMS)* in the ninth edition of Java, these research efforts are not customized for component-based Java projects. To that end, we propose two static component-based RTS approaches called CORTS and its variant C2RTS tailored for component-based Java software systems. CORTS leverages the architectural information such as components and ports, specified in the module descriptor files, to construct module-level dependency graph and identify relevant tests. The variant, C2RTS, is a hybrid approach in which it integrates analysis at both the module and class levels, employing module descriptor files and compile-time information to construct the dependency graph and identify relevant tests.

We evaluated CORTS and C2RTS on 1200 revisions of 12 real-world open source software systems, and compared the results with those of class-level dynamic (Ekstazi) and static (STARTS) RTS approaches. The results showed that CORTS and C2RTS outperformed the static class-level RTS in terms of safety violation that measures to what extent an RTS technique misses test cases that should be selected. Using Ekstazi as the baseline, the average safety violation with respect to Ekstazi was 1.14% for CORTS, 2.21% for C2RTS, and 3.19% for STARTS. On the other hand, the results showed that CORTS and C2RTS selected more test cases than Ekstazi and STARTS. The average reduction in test suite size was 22.78% for CORTS and 43.47% for C2RTS comparing to the 68.48% for STARTS and 84.21% for Ekstazi. For all the studied subjects, CORTS and C2RTS reduced the size of the static dependency graphs compared to those generated by static class-level RTS, leading to faster graph construction and analysis for test case selection. Additionally, CORTS and C2RTS achieved reductions in overall end-to-end regression testing time compared to the retest-all strategy.

1. Introduction

Regression testing is the process of running the existing test cases on a new version of a software system to ensure that the performed modifications do not introduce new faults to previously tested code [1–3]. Regression testing is one of the most expensive activities performed during the lifecycle of a software system with some studies [4–10] estimating that it can take up to 80% of the testing budget and up to 50% of the software maintenance cost. For instance, Google reported that their regression-testing system, TAP [11], experienced a linear growth in both the number of software changes and the average test-suite execution time, which ultimately resulted in a quadratic rise in the overall test-suite execution time. This rapid increase poses a challenge to manage, even for a company with extensive computing resources [12]. Regression test selection (RTS) approaches are used to improve regression testing efficiency [3,12]. RTS is defined as the activity of selecting a subset of test cases from an existing test set to verify that the affected functionality of a program is still correct [3,12,13].

The RTS problem has been studied for over three decades [14,15]. Traditional code-based RTS approaches take four inputs: the two versions (new and old) of a software system, the original test suite, and dependency information of the test cases on the old version. The output

* Corresponding author.
E-mail addresses: mnalrefai@just.edu.jo (M. Al-Refai), m-hammad@just.edu.jo (M.M. Hammad).

https://doi.org/10.1016/j.sysarc.2025.103343
Received 30 May 2024; Received in revised form 12 January 2025; Accepted 12 January 2025
Available online 18 January 2025

is the subset of test cases – from an existing test set – that must be re-executed on the modified version of the software system [12].

RTS techniques vary in the granularity at which they compute test dependencies from test cases to code statements, basic blocks, methods, or classes. Recently, researchers showed that, for individual projects, class-level RTS can be more efficient and beneficial than identifying changes and computing dependencies at lower granularities, e.g., statement and method levels [12,16,17]. Therefore, the current trend [12,16–20] is to focus on class-level RTS by (1) identifying changes at the class level and (2) computing dependencies from test cases to the classes under test. In addition to supporting class-level RTS, these approaches consider a test class as a test case, and thus, select test classes instead of test methods [12,18,19].[1]

Class-level RTS can be static or dynamic, by analyzing dependencies from test cases to classes under test statically or dynamically. A recent extensive experimental evaluation of static class-level RTS [17,18] showed that it is comparable with the state-of-the-art dynamic class-level RTS approach, called Ekstazi [12]. While such a dynamic RTS approach requires code instrumentation and runtime information to find affected tests, static class-level RTS does not require such information, and instead, it builds a dependency graph of program types based on compile-time information, and selects test cases that can reach changed types in the transitive closure of the dependency graph [17,18]. However, static class-level RTS approaches can be unsafe, which means they might miss selecting test cases that are impacted by code changes. The use of Java reflection is the main cause of unsafety in static RTS approaches when compared with dynamic RTS approaches. Reflection in Java allows for runtime behaviors that can be challenging to predict statically, which means static RTS might miss identifying some dependencies during test selection [17,18].

The previous dynamic and static class-level RTS techniques have primarily focused on Java object-oriented projects, without addressing the unique needs of modularized Java applications. With the introduction of the *Java Platform Module System (JPMS)* [21] in Java 9 and newer versions, existing RTS research approaches have not been adapted to accommodate the architectural constructs of component-based Java projects. To bridge this gap, we propose two static compo nent-based RTS approaches, CORTS and its variant C2RTS, specifically designed for component-based Java software systems that are developed using the JPMS architectural constructs.

JPMS provides explicit implementation-level support for well-known architectural constructs, such as components (called *modules*) and ports (called *module directives*). These constructs provide a higher level of abstraction than Java packages and classes. CORTS leverages the architectural constructs information, such as components and ports, presented in the module descriptor files, named *"module-info.java"* [21], to construct module-level dependency graph. The variant, C2RTS, is a hybrid technique that integrates module- and class-level analysis, and therefore, uses both the module descriptor files and part of compile-time information to construct the dependency graph. The two approaches, CORTS and C2RTS, find relevant test cases that can reach some changed module/class in the transitive closure of the dependency graph. Similar to recent RTS approaches [12,16–20], CORTS and C2RTS consider each test class as a test case.

CORTS and C2RTS can improve safety over traditional static class-level RTS techniques by capturing runtime module-level dependencies that are related to reflection and dynamic class loading mechanisms. This is possible because such dependencies are explicitly defined inside the module descriptor files using the `open` and `opens with` directives [21].

We evaluated CORTS and C2RTS in terms of (1) safety and precision violations, (2) reduction in test suite size, (3) reduction in dependency graph size with respect to static class-level RTS techniques, (4) execution time required to construct and analyze the static dependency graph to select relevant test cases, and (5) reduction in the end-to-end regressing testing time compared to the retest-all strategy. We compared the results obtained by CORTS and C2RTS with those of the state-of-the-art class-level dynamic (Ekstazi [12]) and static (STARTS [18]) RTS approaches, using 1200 revisions of 12 real world Maven-based Java software systems.

This paper is organized as follows. Section 2 provides an illustrative example to explain the work of our approach. Section 3 describes the proposed approaches, CORTS and C2RTS. Section 4 presents the evaluation. Section 5 describes the threats to the validity of our approach and results. Related work is summarized in Section 6. Conclusions and plans for future work are outlined in Section 7.

2. Illustrative example

This section presents an illustrative example of a Java 9 Component-Based (CB) application of a university system, which is adapted from the example used in Hammad et al. [22]. We use this example in the following section (i.e., Section 3) to demonstrate how our approaches, CORTS and C2RTS, are used with a CB application.

The university system example is developed according to the Java Platform Module System (JPMS) [21], which is a key feature of project Jigsaw [23], designed to provide a scalable module system for Java. It enables developers to build applications using modular constructs, i.e., components (*modules*) and ports (*module directives*), offering a higher level of abstraction than packages or classes. The modularized Java 9 JRE allows applications to depend on specific modules of the JRE rather than the entire runtime environment. Each module in JPMS includes a descriptor file called "module-info.java", which specifies its dependencies and exported services. The JPMS supports various ports that enable a module to export its services or require services from other modules, facilitating clear and maintainable module interactions.

Fig. 1 shows the component-based architecture of the university system. It is important to mention that Hammad et al. [22] created this university system by converting its equivalent Java 8 Object Oriented version to the CB version according to the OO2CB tool proposed in [22], which is a tool that converts Java 8 OO apps to equivalent Java 9 CB apps following the least-privilege security principle. A *least-privilege architecture* is an architecture in which each component is only granted the exact privileges, in terms of inter-component communications as well as the required JRE modules, it needs to provide its functionality [22,24]. This principle is also important to perform safe and precise regression test selection based on the exact needed inter-component communications/dependencies.

Before presenting the example details, it is also important to mention that generally, there are two common methods for organizing test cases in CB applications: (1) placing them in separate *test-components* or (2) alongside core application classes within *app-components*. The first method, adopted in this paper for illustration, creates distinct components for test classes, aligning with the separation of concerns principle by isolating production and test code. This approach ensures clear boundaries and flexible management of dependencies specific to testing. Both CORTS and C2RTS are compatible with either method. In this paper, we use *test-components* to refer to the modules containing test classes, and use *app-components* to refer to the modules containing the core application classes, i.e., production code.

As depicted in Fig. 1, the university system consists of four *app-components*, i.e., modules,[2] which are `location`, `registration`, `stuService`, and `serviceProvider`. In addition, the system contains three *test-components*, which are `locationTest`, `registrationTest`, and `serviceProviderTest`. The `java.logging` component is also used by the system. The Java classes in the system

[1] From this point until the end of the paper we use the term test case to refer to a test class.

[2] In the paper, we use the terms module and component interchangeably.

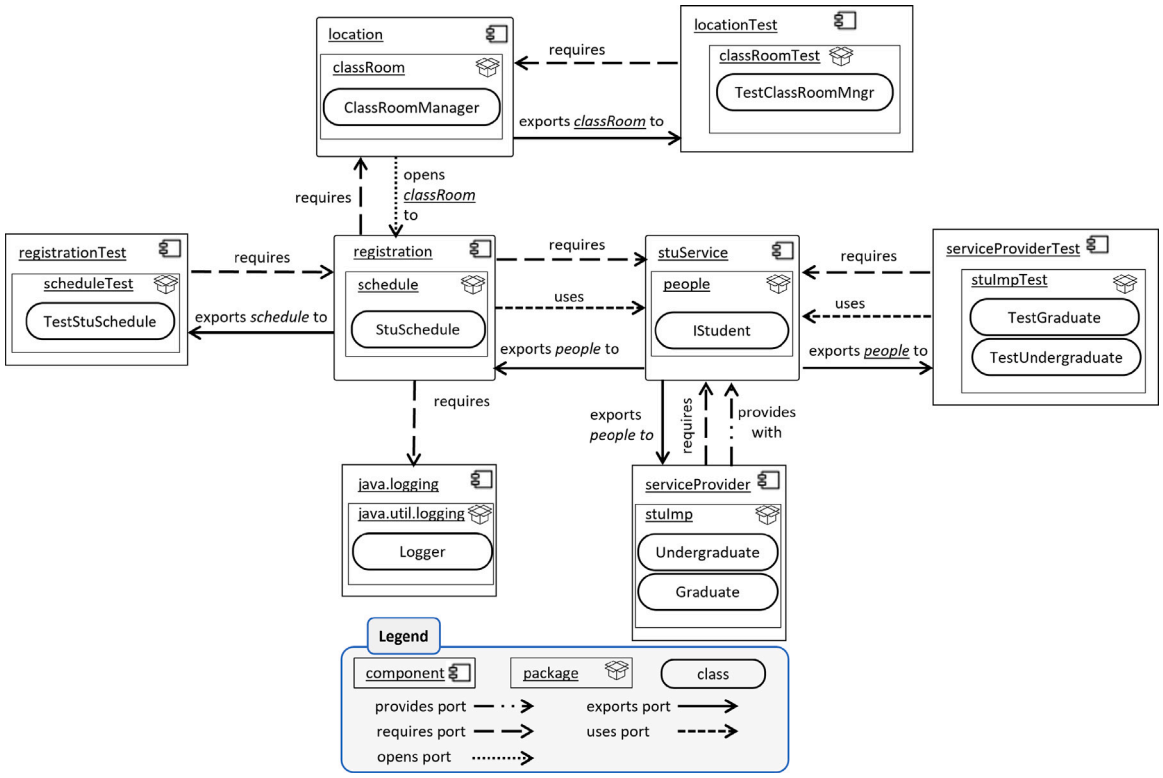


Fig. 1. Component-Based (CB) application adapted from [22].

interact as follows: The `StuSchedule` class generates a suggested schedule for a student and logs relevant details to a log file using the `java.util.Logger` class. Additionally, `StuSchedule` dynamically loads the `ClassRoomManager` class and invokes its methods via Java reflection to retrieve classroom information, which it logs in the students' schedules. A student can either be an `Undergraduate` or a `Graduate`, with both classes implementing the `IStudent` interface.

The corresponding "module-info.java" files for the four *app-components* are presented in Fig. 2(a), while those for the three *test-components* are shown in Fig. 2(b). In the remainder of this section, we discuss some of the key directives used in these "module-info.java" files: `provides with`, `exports to`, `opens to`, and `uses`.

As shown in Fig. 1, the `stuService` component contains the `IStudent` interface inside the `people` package. In order for the `Undergraduate` and `Graduate` classes from the `serviceProvider` component to implement the interface, the `stuService` needs to export the `people` package using the `exports to` port that is shown in Line 12 of Fig. 2(a). In addition to the `exports to` port, the `serviceProvider` component needs to define two more ports. One port to require the `stuService` component as shown in Line 18 of Fig. 2(a) and another `provides with` port to provide the functionalities of the `IStudent` interface using the `Graduate` and `Undergraduate` implementation as shown in Lines 19–21 of Fig. 2(a).

The class `StuSchedule` located in the `registration` component contains a code to dynamically load the class `ClassRoomManager` and invoke its methods using Java reflection. Therefore, the `location` component that contains the `ClassRoomManager` class defines an `opens to` port to open the package `ClassRoom` to the `registration` component as shown in Line 26 of Fig. 2(a). This port enables the classes of the `registration` component to load and access all classes of the `ClassRoom` package using the Java reflection mechanisms.

The test classes `TestGraduate` and `TestUndergraduate`, belonging to the `serviceProviderTest` test-component, use the `IStudent` interface from the `stuService` component. As a result, `serviceProviderTest` declares a `requires` directive in its "module-info.java" file to establish this communication, as shown on Line 8 of Fig. 2(b). Simultaneously, the `stuService` component defines an `exports to` directive to expose the `people` package to `serviceProviderTest`, as illustrated on Line 13 of Fig. 2(a). Additionally, because `IStudent` is an interface, `serviceProviderTest` also declares a `uses` directive, as indicated on Line 9 of Fig. 2(b).

3. Approach

This section describes our proposed component-based RTS approaches, CORTS and C2RTS, which are static analysis tools and based on analyzing dependencies from test cases to the components of the software application under test. CORTS and C2RTS assume that the software application is component-based Java application. It also worth mentioning that if the app is constructed according to the least-privilege architecture, in which each component is only granted the precise dependencies to components and resources that are needed to provide its functionality, our RTS approaches yield more precise test case selection.

Consistent with the current trend in code-based RTS research [12, 18,19], CORTS and C2RTS consider a test class to be a test case. They support both unit and system test cases. The inputs to CORTS and C2RTS are the previous version of the Java application along with its test cases, i.e., the application before modification, and the current (modified) version of the Java application. The output is the set of selected test cases that must be re-executed on the current version of the application.

We present CORTS in Section 3.1 and its variant C2RTS is described in Section 3.2.

3.1. The corts approach

CORTS takes the previous version of a CB Java app along with its test cases, then it parses the module descriptor files (the `module-info.java` files) of all app-components and test-components. While

```
// module-info.java for registration
module registration {
    requires stuService;
    uses people.Istudent;
    requires java.logging;
    exports schedule to registrationTest;
}

// module-info.java for stuService
module stuService {
    exports people to registration;
    exports people to serviceProvider;
    exports people to serviceProviderTest;
}

// module-info.java for serviceProvider
module serviceProvider {
    requires stuService;
    provides people.IStudent
        with stuImp.Undergraduate,
             stuImp.Graduate;
}

// module-info.java for location
module location {
    opens classRoom to registration;
    exports classRoom to locationTest;
}
```

(a) module-info.java files for app-components

```
// module-info.java for registrationTest
module registrationTest {
    requires registration;
}

// module-info.java for serviceProviderTest
module serviceProviderTest {
    requires stuService;
    uses people.Istudent;
}

// module-info.java for locationTest
module locationTest {
    requires location;
}
```

(b) module-info.java files for test-components

Fig. 2. module-info.java files.

parsing the descriptors, CORTS constructs a directed graph, called *Entity Dependency Graph* (EDG), where each node represents a component or a test case (test class), and the directed edges among the nodes represent the various types of dependencies among the components, such as `requires`, `uses` and `provides with` dependencies. After that, CORTS compares the previous version of the CB app with the current version of the app to identify the modified components and flag their corresponding nodes in the EDG. Then, CORTS finds and returns the set of affected test cases that directly or transitively reach a modified component in the EDG. The detailed process of CORTS consists of the three steps:

1. Building the EDG from the component-based application (Section 3.1.1).
2. Identifying the modified components in the EDG (Section 3.1.2).
3. Selecting the affected test cases (Section 3.1.3).

We demonstrate these steps in light of the illustrative example shown in Fig. 1.

3.1.1. Building the EDG from the component-based application

In this step, CORTS parses the `module-info.java` descriptors of the app- and test-components of the previous version of the Java application. While parsing the descriptor files, CORTS builds the EDG, where each node in this directed graph represents a component or a test case, and the directed edges among the nodes represent the various types of dependencies among the components. As an example, Fig. 3 shows the extracted EDG for the CB example shown in Fig. 1.

CORTS distinguishes between descriptor files of app-components and those of test-components. If the `module-info.java` descriptor is for an app-component `A`, then a node is added in the EDG for `A`, and all communication ports that are directed towards or emanating from `A`, e.g., `requires` or `use` ports, are represented in the EDG as directed edges leading to or originating from the node `A`. However, if the descriptor is for a test-component `T`, then CORTS adds a node in the EDG for each individual test class that belongs to `T`. Subsequently, all communication ports that are directed towards or emanating from `T` are represented in the EDG as directed edges leading to or originating from every node representing a test class belonging `T`. CORTS is capable of identifying all test classes associated with a given test-component through a straightforward method. This involves navigating the file system directory designated for the test-component and locating the class files contained inside it. In the context of component-based Java applications organized using JPMS features, each component is assigned a distinct OS directory. This directory houses all Java packages, classes, and the `module-info.java` file pertinent to the component, facilitating the identification process.

When CORTS scans the `module-info.java` file of each component in the CB app, it adds directed edges in the EDG according to the following rules. We demonstrate each of these rules using the extracted EDG shown in Fig. 3 for the illustrative CB example depicted in Fig. 1.

Rule 1 (*Requires Port*). *Let M_1 be a component that requires another component M_2, where this communication is represented using the statement "`requires` M_2" in the `module-info.java` file of M_1. This `requires` port means that a class(es) that belongs to M_1 depends/communicates with a class(es) that belongs to M_2. According to this dependency, CORTS adds a directed edge from node M_1 to node M_2 in the EDG.*

For example, the `registration` component `requires` the `stuService` component as specified in the corresponding module descriptor file shown in Fig. 2, and therefore, a directed edge is added in the EDG from node `registration` to node `stuService` as depicted in Fig. 3. Moreover, as shown in Fig. 2, the `serviceProviderTest` component `requires` the `stuService` component. Therefore, a directed edge is added in the EDG from every test class node belonging to `serviceProviderTest`, i.e., the test classes `TestGraduate` and `TestUndergraduate`, to the node `stuService`, as shown in Fig. 3.

Rule 2 (*Provides With and Uses Ports*). *Let C_1 be a class in module M_1 and A_2 be an abstract class or an interface in module M_2, where*

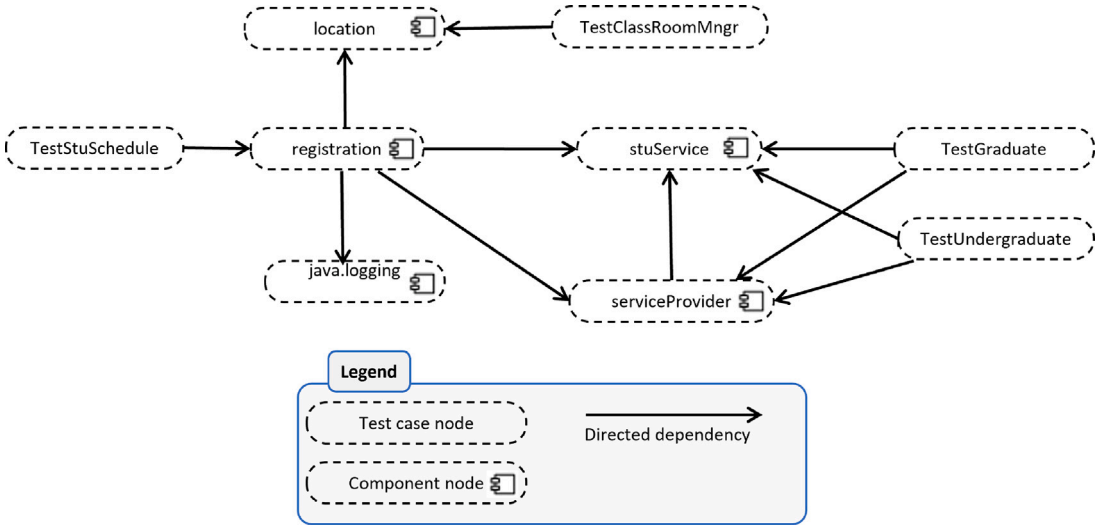


Fig. 3. Entity Dependency Graph (EDG) extracted by CORTS.

C_1 implements or extends A_2. This dependency is represented using the statement "`provides` A_2 `with` C_1" in the `module-info.java` file of M_1. Additionally, let C_3 be a class that belongs to module M_3, where C_3 uses A_2, which is represented using the statement "`uses` A_2" in the `module-info.java` file of M_3. Then, the component M_3 can utilize the `java.util.ServiceLoader` from the `java.base` JPMS JDK module to load implementations (i.e., C_1 belonging to M_1) of the service A_2. According to this dependency from component M_3 to component M_1 that contains the concrete class C_1, CORTS adds a directed edge from node M_3 to node M_1 in the EDG.

For example, as depicted in the module configuration files shown in Fig. 2, the `serviceProvider` component `provides` the interface `IStudent` of the component `stuService` with the concrete classes `Graduate` and `Undergraduate`. Additionally, the `registration` component `uses` the `IStudent` interface. Those communication ports enable the component `registration` to access the component `serviceProvider` and load the two concrete classes, `Graduate` and `Undergraduate`, via the class `java.util.ServiceLoader`. Therefore, a directed edge is added in the EDG from node `registration` to node `serviceProvider` as shown in Fig. 3. Likewise, the test component `serviceProviderTest` `uses` the `IStudent` interface as depicted in Fig. 2, which grants this test component an access to the concrete classes `Graduate` and `Undergraduate` of the component `serviceProvider`. Therefore, a directed edge is added in the EDG from each test class node (i.e., nodes representing `TestGraduate` and `TestUndergraduate` that belong to `serviceProviderTest`) to node `serviceProvider`, as shown in Fig. 3.

Rule 3 (*Opens with Port*). *Let p_1 be a package that belongs to a module M_1, and let this module opens p_1 to another module M_2, such that this dependency is represented using the statement "`opens` p_1 `to` M_2" in the `module-info.java` file of M_1. Then, M_2 can communicate with M_1 and load and access classes of the package p_1 via the Java reflection and dynamic class loading mechanisms. According to this dependency from M_2 to M_1, CORTS adds a directed edge from node M_2 to node M_1 in the EDG.*

For example, in the module configuration files shown in Fig. 2, the `location` component `opens` its package `classRoom` to the `registration` component. Therefore, a directed edge is added in the EDG from node `registration` to node `location`, as shown in Fig. 3.

3.1.2. Identifying the modified components in the EDG

This step involves identifying the modified components to mark their associated nodes in the EDG as modified. CORTS considers a component modified if any of its classes have undergone changes. There are several methods to determine which classes have been modified. For instance, the *Linux diff* command can be used to compare the directories of a component across the previous and current versions of the Java application. Should this command highlight a component's directory due to alterations or removal of any class within it, or the addition of new classes into it, CORTS will then mark the node representing that component in the EDG as modified. Another method involves comparing the smart checksums of the previous and current versions of each compiled Java file (i.e., `.class` files) to identify changed classes [12]. In environments employing Continuous Integration (CI) for Java application development, like `GitHub`, the modifications can also be traced through version control specific commands, such as *git diff*, to find the changed classes and components. Currently, CORTS primarily utilizes the *Linux diff* strategy to pinpoint and mark the modified components within the EDG. However, it is effortless to make CORTS supports other strategies.

For example, if the `ClassRoomManager` class is modified, e.g., some of its source code is changed to add/delete/modify methods, then the component containing this class, which is `location`, is marked as modified in the EDG shown in Fig. 3.

3.1.3. Selecting the affected test cases

In this step, mirroring the methodology of `firewall` static RTS approaches [17,25], CORTS traverses the EDG to identify the nodes of all test cases that reach nodes representing modified components. In particular, CORTS calculates the transitive closure for each test case to find all the components that a test case depends on. Subsequently, the set of impacted test cases whose transitive dependencies include some modified component, is returned as the output by CORTS. We used the `JGraphT` library [26] to construct the EDG and to calculate the transitive closures for the test cases within the EDG.

To complete the demonstration example, if the class `ClassRoomManager` is modified and its component `location` is marked in the EDG shown in Fig. 3, then all test cases that transitively reach the `location` node, which are `TestClassRoomMngr` and `TestStuSchedule`, will be selected and returned as the output of CORTS.

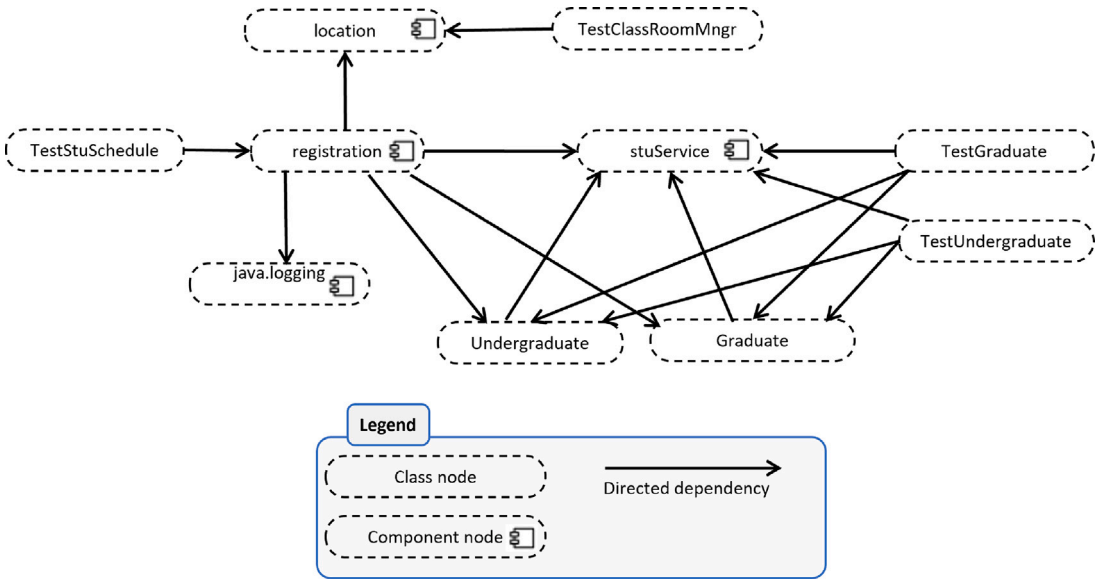


Fig. 4. Entity Dependency Graph (EDG) extracted by C2RTS.

3.2. The c2rts approach

We have developed a hybrid RTS approach that combines aspects from both the **C**omponent-level and **C**lass-level **RTS** techniques, called C2RTS. This variant of CORTS integrates module- and class-level dependency analyses, trading off to strike a balance between safety and precision by adjusting the level of granularity from modules to classes depending on the specific classes where code changes have been made. C2RTS trades off some safety for increased precision compared to CORTS.

While constructing the EDG, C2RTS distinguishes between modified and unmodified app-components within the Java application. Specifically, each unmodified app-component is represented as a single node in the EDG, whereas all classes belonging to a modified app-component are represented as individual nodes.

As an example for an EDG constructed by C2RTS, Fig. 4 shows the constructed EDG given that the app-component `serviceProvider` is identified as a modified app-component by C2RTS, and thus, all classes belonging to this app-component are represented as individual nodes in the EDG. The remaining app-components are unmodified, and thus, each of them is represented as a single node in the EDG. The subsequent subsections elaborate on the entire process undertaken by C2RTS to construct the EDG and select test cases.

3.2.1. Building the EDG from the component-based application

We explain the steps applied by C2RTS to build the EDG nodes and edges from the component-based application.

3.2.1.1. Mappings from components to nodes in the edg. This section explains how C2RTS maps the unmodified app-components, modified app-components, and test-components to nodes in the EDG.

Representing modified app-components as nodes in the EDG. Given the previous and current versions of the CB app, if an app-component is modified between the two versions, then instead of representing this app-component as a single node in the EDG, C2RTS represents each class belonging to the app-component as a single node in the EDG.

In our illustrative example, we suppose that the app-component `serviceProvider` depicted in Fig. 1 is identified as modified by C2RTS. Consequently, all the classes of this component (i.e., the `Undergraduate` and `Graduate` classes) are represented as nodes in the EDG, as shown in the EDG represented in Fig. 4.

Representing unmodified app-components as nodes in the EDG. The unmodified app-components of the application are handled using the same way employed by CORTS, where they are presented as nodes in the EDG using the same method described previously in Section 3.1.1. For example, the app-component `location` is identified by C2RTS as unmodified, and therefore, is represented as a single node in the EDG.

Representing test-components as nodes in the EDG. Similar to CORTS, the C2RTS approach creates a separate node for each individual test class in the EDG. For example, the EDG shown in Fig. 4 contains a node for each test class, such as the test classes `TestStuSchedule` and `TestGraduate`.

Next, we describe the various ways of C2RTS for (1) extracting dependencies among classes of a modified app-component in Section 3.2.1.2, (2) extracting dependencies among unmodified app-components in Section 3.2.1.3, (3) extracting dependencies between unmodified and modified app-components in Section 3.2.1.4, and (4) extracting dependencies between test-components and app-components in Section 3.2.1.5.

3.2.1.2. Extracting dependencies among modified app-component classes. The dependencies among the classes of a modified app-component are extracted using the Oracle Java Class Dependency Analyzer (`jdeps`) tool [27].[3] These dependencies are represented as directed edges in the EDG between the nodes representing the classes of the modified app-component.

3.2.1.3. Extracting dependencies among unmodified app-components. The dependencies among the unmodified app-components are extracted and represented in the EDG according to Rules 1, 2, and 3 described previously in Section 3.1.1. For example, in the EDG represented in Fig. 4, C2RTS added an edge from node `registration` to node `location` according to Rule 3.

3.2.1.4. Extracting dependencies between unmodified and modified app-components. The dependencies between the unmodified components and the classes of the modified components are extracted using: (1) information extracted from the component configuration files, the `module-info.java` files, and (2) information extracted using the `jdeps` tool. These extracted dependencies are used to construct the

[3] jdeps now is part of the standard Java library, and is used to analyze the module-level, package-level, and class-level dependencies of Java class files.

EDG according to three formally defined rules, Rules 4, 5, 6. The three rules are described using the following assumptions:

- Let *App* be a previous version of a component-based Java application that was modified to the current version *App′*.
- Let a module M_1 represents an app-component that was identified as modified between *App* and *App′*, i.e., some classes that belong to M_1 were modified.
- Let a module M_2 represents an app-component that belongs to *App* and this module is not modified in *App′*, i.e., unmodified app-component.

Given these assumptions, C2RTS represents M_2 as a single node in the EDG, and instead of representing M_1 as a single node, C2RTS represents all the classes/interfaces belonging to M_1 as nodes in the EDG. Subsequently, C2RTS extracts the dependencies between the classes of M_1 and the module M_2 and reflects them as edges in the EDG based on the following rules:

Rule 4 (*Provides with and Uses Ports*). *Let C_1 be a class in module M_1 and A_2 be an abstract class or an interface in module M_2, where C_1 implements or extends A_2. This port is represented using the statement "`provides` A_2 `with` C_1" in the "`module-info.java`" file for M_1. Additionally, let C_3 be a class that belongs to an unmodified module M_3 (M_3 is an app-component), where C_3 uses A_2, and this port is represented using the statement "`uses` A_2" in the "`module-info.java`" file for M_3. Then, the component M_3 can utilize the `java.util.ServiceLoader` from the `java.base` JPMS JDK module to load the implementation of C_1 that belongs to M_1. According to this dependency from component M_3 to class C_1, C2RTS adds a directed edge from node M_3 to node C_1 in the EDG.*

We explain how this rule is applied to the EDG nodes represented in Fig. 4 given the module configuration files shown in Fig. 2. In the configuration files, the `serviceProvider` component `provides` the interface `IStudent` of the `stuService` component `with` the concrete classes `Undergraduate` and `Graduate`. Additionally, the `registration` component `uses` the `IStudent` interface. These ports enable the component `registration` to load implementations of the two concrete classes `Graduate` and `Undergraduate`. Therefore, two directed edges are added in the EDG from the node `registration` to the nodes `Graduate` and `Undergraduate` as shown in Fig. 4.

Rule 5 (*Requires Port from Unmodified to Modified App-Component*). *Let M_2 requires M_1, where this port is represented using the statement "`requires` M_1" in the "`module-info.java`" file for M_2. Then, according to this dependency from an unmodified module (M_2) to a modified module (M_1), C2RTS uses the `jdeps` technique with M_2 and M_1 to find the set of dependencies from classes belonging to M_2 into classes belonging to M_1. Let `jdeps` result included some dependencies from some class(es) of M_2 to a class called C_1 that belongs to M_1. Then, C2RTS adds a directed edge from node M_2 to node C_1 in the EDG.*

Rule 6 (*Requires Port from Modified to Unmodified App-Component*). *Let M_1 requires M_2, which is represented using the statement "`requires` M_2" in the "`module-info.java`" file of M_1. According to this `require` dependency from a modified module (M_1) to an unmodified module (M_2), C2RTS applies the `jdeps` technique with M_1 and M_2 to find the classes of M_1 that depend on classes of M_2. Let `jdeps` result included that a class C_1 belonging to M_1 depends on some class(es) belonging to M_2. Then, C2RTS adds a directed edge from node C_1 to node M_2 in the EDG.*

For example, in the modules' configurations shown in Fig. 2, the app-component `serviceProvider`, which was identified by C2RTS as modified, `requires` the unmodified app-component `stuService`. Hence, the `jdeps` technique is applied with these two components and returns that the classes `Undergraduate` and `Graduate` belonging to `serviceProvider` depend on the class `ISudent` that belongs to `stuService`. Consequently, directed edges are added in the EDG from the nodes representing `Undergraduate` and `Graduate` to the node representing `stuService`, as shown in Fig. 4.

3.2.1.5. Extracting dependencies between test-components and app-components. The C2RTS approach represents each individual test class belonging to a test-component as a single node in the EDG. The dependencies from test classes to unmodified app-components are extracted and represented as edges in the EDG according to Rules 1, 2, and 3 explained previously in Section 3.1.1. On the other hand, dependencies from test classes to classes belonging to modified app-components are extracted according to the following two rules, Rules 7 and 8 which are modified versions of Rules 4 and 5, respectively.

Rule 7 (*Provides with and Uses Ports*). *Let C_1 be a class in module M_1 and A_2 be an abstract class or an interface in module M_2, where C_1 implements or extends A_2. This port is represented using the statement "`provides` A_2 `with` C_1" in the "`module-info.java`" file for M_1. Additionally, let TM_3 be a test-module, where some test classes belonging to TM_3 use A_2, and this dependency is represented using the statement "`uses` A_2" in the "`module-info.java`" file of TM_3. This `uses` port enables test classes belonging to the test-component TM_3 to utilize the `java.util.ServiceLoader` from the `java.base` JPMS JDK module to load the implementation of C_1 that belongs to M_1. Subsequently, C2RTS applies the `jdeps` technique with TM_3 and M_2 to find the test classes of TM_3 that depend on A_2. Let `jdeps` returned that a test class TC_3 belonging to TM_3 depends on A_2. Then, C2RTS adds a directed edge from node TC_3 to node C_1 in the EDG because TC_3 can load C_1 via the class `java.util.ServiceLoader`.*

For example, in the module configuration files shown in Fig. 2, the `serviceProvider` component `provides` the interface `IStudent` of the `stuService` component `with` the concrete classes `Undergraduate` and `Graduate`. Additionally, the test-component `serviceProviderTest` `uses` the `IStudent` interface as depicted in Fig. 2. Therefore, C2RTS finds, using `jdeps`, which test classes of `serviceProviderTest` depend on `IStudent`, and the `jdeps` returns that the test classes `TestGraduate` and `TestUndergraduate` depend on `IStudent`. Subsequently, directed edges are added in the EDG from each of these test classes to the concrete classes `Graduate` and `Undergraduate` as depicted in Fig. 4.

Rule 8 (*Requires Port from Test-Component to Modified App-Component*). *Let TM_1 be a test-component that requires the modified app-component M_1, such that this dependency is represented using the statement "`requires` M_1" in the "`module-info.java`" file for TM_1. Then, based on this dependency, C2RTS uses the `jdeps` technique with TM_1 and M_1 to find the set of dependencies from test classes belonging to TM_1 into classes belonging to M_1. From these dependencies, C2RTS extracts the names of the source and target classes and connect their corresponding nodes in the EDG with the proper directed edges.*

3.2.2. Mark modified nodes and select affected test cases

To mark the modified classes and compute the set of selected test cases, C2RTS applies the same steps explained previously in Sections 3.1.2 and 3.1.3 with one difference. That is instead of marking nodes representing modified components in the EDG, C2RTS marks the nodes that represent modified classes. Then, C2RTS computes the transitive closure of each test case to find all components and classes that each test depends on. Thereafter, the set of impacted test cases whose transitive dependencies in the EDG include some changed type, is returned by C2RTS as the output. For example, if the class `Undergraduate` is modified and marked in the EDG shown in Fig. 4, then the test cases `TestUndergraduate`, `TestGraduate`, and `TestStuSchedule` will be selected and returned as the output of C2RTS.

4. Experimental evaluation

The goal of the evaluation is to compare CORTS and C2RTS with the *state-of-the-art* class-level RTS tools in terms of (1) safety violation, (2) precision violation, (3) test suite reduction, (4) size of the dependency graph that represents static dependencies from test cases to code entities, and (5) reduction in test selection and execution times. An RTS technique is safe if it does not miss any modification-traversing test cases that should be selected for regression testing. An RTS technique is precise if it does not select non-modification traversing test cases. A test case is considered as a modification-traversing test case if it exercises during its execution a modified, new, or previously removed code statements. Only modification-traversing test cases can reveal faults in the modified version of a software system, and hence, must be selected for regression testing.

We compared CORTS and C2RTS with two RTS tools, Ekstazi [12] and STARTS [18]. They are both *state-of-the-art* for class-level RTS and have been widely evaluated on a large number of revisions of real world projects [17]. The class-level RTS process identifies changes at the class level, instead of method or statement levels, and selects every test-class that traverses or depends on any changed class. Ekstazi uses dynamic analysis and STARTS uses static analysis of compiled Java code. We compared CORTS and C2RTS with these class-level RTS approaches because we aimed to investigate (1) how the safety can be improved by raising the RTS granularity from class-level to component-level, (2) how increasing the RTS granularity from class-level to component-level impacts the precision and test suite reduction, and (3) how increasing the RTS granularity reduces the size of the static component-level dependency graph compared to the static class-level dependency graph.

In order to evaluate the safety and precision of CORTS and C2RTS, we computed their safety violations and precision violations w.r.t. Ekstazi [12]. Ekstazi is a code-based RTS approach known to be safe in terms of selecting all the modification-traversing test classes, widely evaluated on a large number of revisions, and being adopted by several popular open source projects; as such it can be considered the state-of-the-art for class-level dynamic RTS tools. Assuming that a program **P**, which has an original test suite **T**, was modified to a new version **P'**. Furthermore, assuming that two RTS approaches, $\mathbf{RTS_1}$ and Ekstazi were applied to select test cases from **T** based on the code modifications to move the program from **P** to **P'**, such that $\mathbf{RTS_1}$ selected the set of test cases $\mathbf{T_{RTS_1}}$ and Ekstazi selected the set of test cases $\mathbf{T_{Ekstazi}}$. Then, the safety violation of $\mathbf{RTS_1}$ w.r.t. Ekstazi, precision violation of $\mathbf{RTS_1}$ w.r.t. Ekstazi, and reduction in test suite size obtained by $\mathbf{RTS_1}$ are defined as follows:

$$SafetyViolation\ w.r.t.\ Ekstazi = \frac{|\mathbf{T_{Ekstazi}} \setminus \mathbf{T_{RTS_1}}|}{|\mathbf{T_{Ekstazi}} \cup \mathbf{T_{RTS_1}}|}$$

$$PrecisionViolation\ w.r.t.\ Ekstazi = \frac{|\mathbf{T_{RTS_1}} \setminus \mathbf{T_{Ekstazi}}|}{|\mathbf{T_{Ekstazi}} \cup \mathbf{T_{RTS_1}}|}$$

$$Test\ suite\ reduction\ obtained\ by\ RTS_1 = \frac{|\mathbf{T}| \setminus |\mathbf{T_{RTS_1}}|}{|\mathbf{T}|}$$

The safety violation, precision violation, and test suite reduction are multiplied by 100 to make them percentages. Lower percentages for safety violation, precision violation, and higher percentages for test suite reduction are better [17,28]. The size of the static dependency graph is computed in terms of number of nodes and edges of the graph.

4.1. Research questions

In this research, we try to answer the following Research Questions (RQ):

- RQ1: What is the safety violation w.r.t. Ekstazi of the proposed static component-level RTS approaches CORTS and C2RTS? Furthermore, do CORTS and C2RTS reduce the safety violation w.r.t. Ekstazi (i.e., improve safety) compared to the static class-level RTS approach STARTS?
- RQ2: What is the precision violation w.r.t. Ekstazi of the proposed approaches CORTS and C2RTS? Furthermore, how does this precision violation compare to the precision violation w.r.t. Ekstazi achieved by the static class-level RTS approach STARTS?
- RQ3: What is the reduction in test suite size achieved by CORTS and C2RTS?
- RQ4: How does the size of the static dependency graphs extracted by CORTS and C2RTS compare to the size of the static dependency graph extracted by STARTS?
- RQ5: What is the time taken by CORTS and C2RTS to construct and analyze the static dependency graph to select relevant tests, and what is their overall end-to-end testing time?

4.2. Subjects

We evaluated CORTS and C2RTS using the 12 subjects listed in Table 1. These are open-source real-world Java projects, which are known to be compatible with Ekstazi and STARTS since they were widely used in their evaluation [12,17,18]. Table 1 shows for each subject, the latest revision (i.e., most recent revision of the project) on which our experiments started (**SHA**), the number of the source classes (**CLASSES**) of the latest reversion, i.e., classes of the core program without counting test classes, the number of the source test classes (**TESTS**) of the latest reversion, number of recovered components of the latest revision (**COMPS**), number of ports between the recovered components (**PORTS**), and the number of used revisions (**REVS**).

Converting the projects to equivalent component-based projects. It was not possible to evaluate CORTS and C2RTS using existing open-source component-based Java applications, i.e., multi-module Java applications developed using the JPMS capabilities. There are two main reasons for that. First, the great majority of existing open-source Object Oriented (OO) Java applications have not been converted to component-based equivalent applications using JPMS. For example, as mentioned in Hammad et al. [22] after analyzing more than 1300 open-source Java projects, they found that only 33 are utilizing JPMS capabilities. This finding comports with the results reported in prior work [29] as well. Second, even for the 33 existing component-based projects that utilize JPMS capabilities, the modules of each project are open to all the system, leading to a situation in which components (i.e., modules) are granted more access than they need to function, and this violates the least-privilege architecture principle. Additionally, these projects are relatively small in size, significantly smaller than those listed in Table 1, and were created for educational purposes, meaning they are not real-world component-based Java applications. Therefore, we could not use these component-based projects to evaluate CORTS and C2RTS.

In order to overcome this challenge, we converted the OO Java projects listed in Table 1 to equivalent component-based Java projects. To do that, we leveraged the `O2CB` [22] which utilizes the JPMS capabilities and converts an OO Java application to an equivalent component-based Java application according to the least-privilege security principle. The `O2CB` uses a component recovery framework implemented by Garcia et al. [30], called ARCADE, to automatically determine the components of an OO application. The ARCADE framework utilizes several well-known component recovery tools such as Architecture Recovery using Concerns (ARC) [31], Bunch [32], and Algorithm for Comprehension-Driven Clustering (ACDC) [33]. `O2CB` [22] takes as inputs the suggested components provided by the ACDC tool and the binary code of the OO Java application, and outputs the equivalent component-based Java application that utilizes the JPMS features along with all the modules' descriptors, i.e., the "`module-info.java`" files, generated according to the least-privilege security principle.

Selecting revisions. We downloaded the revisions of every subject among the 12 subjects listed in Table 1 using the methodology in Legunsen et al. [17]. First, we found the latest revision (specified by **SHA**

Table 1
The Java projects used in our study.

Subject	SHA	CLASSES	TESTS	COMPS	PORTS	REVS
commons-math	96f2b16	864	485	59	1116	100
commons-configuration	5de7c48	261	171	18	190	100
commons-compress	a189697	201	105	22	190	100
commons-collections	f9f99cc	351	230	23	251	100
commons-dbcp	23f6717	60	54	4	12	100
commons-io	8d1b994	128	106	9	53	100
commons-lang	82fd251	154	153	13	83	100
commons-validator	e2edf6a	64	76	3	6	100
commons-pool	fde71c6	48	26	5	11	100
JFreeChart	86abdc8	638	344	33	443	100
jankotek.mapdb	a333530	87	61	11	43	100
OpenTripPlanner	45c1a9f	1099	285	147	2724	100

Table 2
Average and median safety violation w.r.t. Ekstazi.

Subject	A-SV_{CORTS} %	M-SV_{CORTS} %	A-SV_{C2RTS} %	M-SV_{C2RTS} %	A-SV_{STARTS} %	M-SV_{STARTS} %
commons-math	**0.0**	0.0	0.04	0.0	0.58	0.0
commons-configuration	**11.53**	0.0	19.26	0.0	22.04	1.89
commons-compress	**0.0**	0.0	**0.0**	0.0	**0.0**	0.0
commons-collections	**0.0**	0.0	**0.0**	0.0	**0.0**	0.0
commons-dbcp	**0.0**	0.0	0.11	0.0	0.56	0.0
commons-io	**0.0**	0.0	**0.0**	0.0	**0.0**	0.0
commons-lang	**0.0**	0.0	**0.0**	0.0	**0.0**	0.0
commons-validator	**0.14**	0.0	1.57	0.0	6.19	0.0
commons-pool	**0.79**	0.0	1.29	0.0	1.65	0.0
JFreeChart	**0.0**	0.0	**0.0**	0.0	**0.0**	0.0
jankotek.mapdb	**0.0**	0.0	3.04	2.12	3.32	3.22
OpenTripPlanner	**1.12**	0.0	1.21	0.0	3.97	1.36

A- or **M-SV_i** is the average/median (per subject) safety violation of a tool (i.e., CORTS, C2RTS, or STARTS) with respect to Ekstazi.

in Table 1) that satisfied these conditions: (1) does not have a build or compile error, (2) no test case failures, and (3) successfully ran with STARTS and Ekstazi. Second, among all the revisions preceding **SHA**, we selected up to a hundred revisions (including the **SHA** revision) that met these conditions. The total number of selected revisions, for the 12 subjects, was 1200. These revisions met the prerequisites for Ekstazi and STARTS: (1) Maven version 3.2.5 or above, (2) Surefire version 2.14 or above, (3) JUnit version 3 or above, (4) Java version 1.8 or above. We used `OO2CB` [22] to convert each of the 1200 revisions to an equivalent component-based version. Table 1 shows for each subject, the numbers of recovered components (**COMPS**) and ports (**PORTS**) among the components of the latest subject's revision used in our study.

For each subject, starting from the oldest revision, among the hundred revisions, up to the most recent revision specified by **SHA**, we ran Ekstazi and STARTS techniques on the successive pairs of revisions, and ran CORTS and C2RTS on the corresponding component-based versions of these revisions. To identify changed classes between the previous and current pair of revisions, Ekstazi compares the smart checksums of the previous and current versions of each compiled Java file (i.e., .class files). STARTS reuses the part of the Ekstazi source code to compute smart checksums and identify changed classes in the same way. In order to ensure equitable comparisons with both Ekstazi and STARTS, we adhere to the same methodology for comparing smart checksums to identify changed classes, subsequently marking the components housing these classes as modified. In particular, the list of changed classes in STARTS can be generated by executing the Linux command-line `STARTS: diff`,[4] and we utilized this command-line in our experiments to generate the list of classes that are identified as changed through smart checksum comparisons.

The experimental dataset, which comprises the ACDC-recovered architectures for all 1200 revisions of the 12 Java projects, is publicly available at https://github.com/mohammedrefai/RTS_ComponentLevel . Each revision's files are labeled with their corresponding SHA identifier, enabling users to retrieve the exact source code revision directly from the respective project's GitHub repository. The following subsections present and discuss the RTS results of all our experiments.

4.3. RQ1: Safety violation

Table 2 shows for each subject the results of the median and average safety violation w.r.t. Ekstazi achieved by CORTS, C2RTS, and STARTS. The median and average values are computed per subject among all the subject's revisions. As shown in Table 2, CORTS and C2RTS achieved better results for safety violation compared to STARTS.

The median safety violation obtained by CORTS was zero for all the 12 subjects, while C2RTS had a value greater than zero for only one subject. For STARTS, the median safety violation was higher than zero for three subjects, i.e., 1.89% for `commons-configuration` and 3.22 for `jankotek.mapdb`. The average safety violation values of CORTS and C2RTS were smaller than those for STARTS for 7 out of the 12 subjects, while all the RTS approaches achieved an average safety violation of zero for the remaining subjects. As it can be seen in Table 2, CORTS reduced the average safety violation almost by half from 22.04% to 11.53% for `commons-configuration`.

The proposed approaches, CORTS and C2RTS, outperformed STARTS in terms of safety violation because they compute dependencies from test cases to code entities at a higher level of granularity (i.e., component-level) than STARTS. This component-level dependency analysis results in higher over-estimation of test dependencies compared to class-level test dependencies, in which more impacted (i.e., modification-traversing) test cases are selected. In particular, the static analysis of test dependencies at the component (or module) level rather than at the class-level can lead to the identification of a broader set of potentially impacted test cases. This is due to the module-level analysis treating all classes within a module as a single entity. By considering the module as a unified unit, this approach inherently accounts for inter-class interactions within the module including dynamic dependencies that involve reflection, even without explicitly tracking such dynamic dependencies. This holistic view increases the likelihood of capturing

[4] STARTS provides the command-line option to list types identified as changed via smart checksum computation.

Table 3
Average and median precision violation with w.r.t. Ekstazi.

Subject	A-PV$_{CORTS}$ %	M-PV$_{CORTS}$ %	A-PV$_{C2RTS}$ %	M-PV$_{C2RTS}$ %	A-PV$_{STARTS}$ %	M-PV$_{STARTS}$ %
commons-math	83.47	96.1	50.27	53.33	**33.12**	25.0
commons-configuration	59.68	64.57	45.64	59.19	**24.54**	22.14
commons-compress	75.41	86.36	62.54	79.09	**53.37**	64.0
commons-collections	58.54	95.21	18.89	0.0	**7.15**	0.0
commons-dbcp	57.41	61.29	30.05	31.81	**18.18**	3.03
commons-io	53.09	80.19	28.24	0.0	**16.21**	0.0
commons-lang	63.88	84.07	56.97	76.35	**48.16**	63.84
commons-validator	60.11	92.11	40.25	11.21	**17.45**	10.71
commons-pool	55.01	54.54	53.71	52.17	**35.11**	26.66
JFreeChart	49.88	73.17	42.21	0.0	**32.33**	0.0
jankotek.mapdb	70.29	77.38	31.89	20.41	**29.02**	17.74
OpenTripPlanner	87.69	91.42	87.67	91.41	**73.26**	75.0

A- or **M-PV$_i$** is the average/median (per subject) precision violation of a tool (i.e., CORTS, C2RTS, or STARTS) with respect to Ekstazi.

Table 4
Reduction in test suite size.

Subject	A-R$_{CORTS}$ %	A-R$_{C2RTS}$ %	A-R$_{STARTS}$ %	A-R$_{Ekstazi}$ %
commons-math	10.98	48.57	76.27	**89.94**
commons-configuration	18.03	32.51	66.46	**77.03**
commons-compress	11.01	24.07	56.42	**86.42**
commons-collections	36.96	77.96	92.87	**95.51**
commons-dbcp	10.32	41.74	55.98	**67.72**
commons-io	36.64	61.66	82.05	**89.41**
commons-lang	26.19	35.86	53.32	**90.07**
commons-validator	31.52	54.17	90.45	**91.55**
commons-pool	19.77	21.41	50.03	**69.58**
JFreeChart	43.83	51.12	84.64	**93.32**
jankotek.mapdb	8.12	52.56	55.61	**70.29**
OpenTripPlanner	20.04	20.06	57.67	**89.67**

A-R$_X$ is the average reduction (per subject) in test suite size achieved by an RTS approach **X**.

dependencies that might be overlooked when analyzing at the finer granularity of individual classes.

Moreover, the average safety violation values of CORTS are smaller than those of C2RTS. This is because C2RTS mixes tracking dependencies both between modules and within them at the class-level for the modified modules, in which inter-class dynamic dependencies that are related to reflection are missed by C2RTS, resulting in missing impacted test cases that are captured by CORTS.

It is essential to acknowledge that the component recovery tools utilized, namely ACDC and OO2CB, are based on static analysis and do not detect the dynamic class dependencies or communications associated with dynamic class loading and reflection. Consequently, the resultant component-based applications in our experimentation lack the "opens with" directive within the generated "module-info.java" files. Consequently, CORTS and C2RTS overlooked impacted test cases, resulting in safety violation values higher than zero for some of the subjects as seen in Table 2. We anticipate that CORTS and C2RTS will yield diminished safety violation values, potentially zero or near-zero, provided that reflection-related dependencies are comprehensively captured and represented within the "module-info.java" files of the evaluation subjects. This would entail modifications to ACDC and OO2CB to accurately capture and represent reflection-related dependencies within the recovered component-based applications. We plan to investigate this direction in the future.

4.4. RQ2: Precision violation

Table 3 shows, for each subject, the results of the median and average precision violation w.r.t. Ekstazi achieved by CORTS, C2RTS, and STARTS. The median and average values are computed per subject among all the subject's revisions.

The average and median safety violations of CORTS and C2RTS are higher than those of STARTS. This is because CORTS and C2RTS compute test dependencies at a higher levels of granularity than STARTS and have higher overestimation of impacted test cases than that of STARTS. On the other hand, the average/median precision violation values of C2RTS are smaller when compared with those yielded by CORTS with a significant variance observed across most of the subjects. For example, the average and median precision violation values of CORTS are 58.54% and 95.21%, respectively, for the subject `commons-collections`. These values are reduced by C2RTS to 18.89% and 0.0%, respectively.

C2RTS did make more mistakes in choosing irrelevant test cases compared to STARTS, but the precision violation yielded by C2RTS was not too far from that provided by STARTS. For 8 out of the 12 subjects, C2RTS was, on average, only up to 13% less accurate than STARTS. For the remaining subjects, the difference went up to 21%. Interestingly, in 3 out of the 12 subjects, C2RTS had a median precision violation of 0%.

4.5. RQ3: Test suite reduction

Table 4 shows for each subject the average reduction in test suite size achieved by CORTS, C2RTS, STARTS, and Ekstazi. The average values are computed per subject among all the subject's revisions. The four RTS approaches achieved reduction in test suite size. The average reduction in test suite size overall the 12 subjects was 22.78% for CORTS, 43.47% for C2RTS, 68.48% for STARTS, and 84.21% for Ekstazi. The highest reduction was yielded by Ekstazi since it is a dynamic approach.

It is evident that (1) both CORTS and C2RTS achieved a reduction for all the subjects even though they perform RTS at a higher level of granularity than STARTS, and (2) C2RTS increased the reduction compared to CORTS from 22.78% to 43.47% on average since it tracks dependencies within the modified components at the class-level. Moreover, C2RTS achieved high reduction by more than 50% on average for 5 out of the 12 subjects, and a reduction by more than 40% on average for 2 other subjects. Furthermore, the comparative analysis with STARTS reveals that C2RTS maintains a competitive edge, with the difference in average test suite reduction between C2RTS and STARTS not surpassing 21% for 5 subjects and remaining below 38% across all the 12 subjects.

Table 5
Dependency graph size.

Subject	$NODES_{CORTS}$	$EDGES_{CORTS}$	$NODES_{C2RTS}$	$EDGES_{C2RTS}$	$NODES_{STARTS}$	$EDGES_{STARTS}$
commons-math	503	4391	567	5090	2099	12 689
commons-configuration	190	1532	284	2470	827	4743
commons-compress	147	933	219	1713	547	2299
commons-collections	202	1396	236	1763	907	3536
commons-dbcp	36	147	126	684	178	711
commons-io	109	554	154	881	336	1017
commons-lang	167	982	227	1537	746	2252
commons-validator	77	208	142	581	179	592
commons-pool	27	108	124	574	208	748
JFreeChart	373	2594	468	4298	1033	7092
jankotek.mapdb	197	876	494	4600	1281	7342
OpenTripPlanner	432	5335	698	8910	2884	15 479

$NODES_X$ or **$EDGES_X$** is the average number of nodes or edges in the dependency graph that was constructed by an RTS approach ($_X$).

Table 6
Dependency graph size reduction ratios of CORTS and C2RTS with respect to STARTS.

Subject	R_NODES_{CORTS}	R_EDGES_{CORTS}	R_NODES_{C2RTS}	R_EDGES_{C2RTS}
commons-math	4.17	2.89	3.70	2.49
commons-configuration	4.35	3.10	2.91	1.92
commons-compress	3.72	2.46	2.50	1.34
commons-collections	4.49	2.53	3.84	2.01
commons-dbcp	4.94	4.84	1.41	1.04
commons-io	3.08	1.84	2.18	1.15
commons-lang	4.47	2.29	3.29	1.47
commons-validator	2.32	2.59	1.26	1.02
commons-pool	7.56	6.91	1.66	1.31
JFreeChart	2.76	2.73	2.21	1.65
jankotek.mapdb	6.48	8.37	2.59	1.58
OpenTripPlanner	6.67	2.91	4.12	1.73

R_NODES_X/R_EDGES_X is the average reduction ratio of nodes/edges of class-level dependency graph achieved by an RTS approach ($_X$).

4.6. RQ4: Reduction in dependency graph size

Table 5 shows – for each subject – the average number of nodes and edges of the static dependency graphs extracted by CORTS, C2RTS, and STARTS. The average values are computed per subject among all the subject's revisions. It is evident that CORTS and C2RTS generated dependency graphs of smaller sizes compared to STARTS.

Table 6 shows, for each subject, the average size reduction ratio of the dependency graphs extracted by CORTS and C2RTS with respect to the dependency graph extracted by STARTS. The size reduction ratio is computed separately for nodes and edges as follows. For a specific revision of a subject, the size reduction ratio for nodes/edges achieved by CORTS/C2RTS is computed as the number of nodes/edges of the graph extracted by STARTS divided by the number of nodes/edges of the graph extracted by CORTS/C2RTS.

Referring to the data in Table 6, CORTS achieved an average reduction in the STARTS dependency graph node count by a factor starting from 4 up to 7 for 8 of the subjects, and by a factor of approximately 3 for the remaining subjects. On the other hand, C2RTS achieved an average reduction in the STARTS dependency graph node count by a factor higher than 2 (i.e., ranging from 2.18 to 4.12) for 9 subjects out of the 12 subjects. Furthermore, CORTS achieved an average reduction in the STARTS dependency graph edge count by factors ranging approximately from 2 up to 8 for 11 subjects, while C2RTS achieved an average reduction in edge count by factors ranging from 1.02 up to 2.49.

The results presented in Table 6 are encouraging and indicating that CORTS and C2RTS are effective in minimizing the static dependency graph size compared to class-level RTS techniques. This capability is crucial and presents significant implications for several reasons. First, the reduced complexity of dependency graphs makes our RTS approaches more scalable to very large applications such as enterprise-level applications with extensive codebases. Second, smaller graphs require less computational resources for analysis and consume less memory. Furthermore, this efficiency in graph size management is particularly beneficial in cloud-based Continuous Integration (CI) environments, where resource and memory consumption directly influences costs, suggesting that such optimizations can result in economical advantages.

4.7. RQ5: Selection phase and end-to-end testing times

The end-to-end execution time of an RTS approach includes two main phases, which are: (1) the *selection phase* that analyzes what test cases to select, and (2) the *execution phase* that runs the selected test cases. For static RTS approaches, the *selection phase* time consists of the time taken to construct the static dependency graph, read adapted classes and flag them in the graph, and analyze (i.e., traverse) the graph to select relevant test cases. Table 7 reports the *selection phase* time for CORTS (**$SELECT_{CORTS}$**), C2RTS (**$SELECT_{C2RTS}$**), and static class-level RTS (**$SELECT_{STARTS\text{-}like}$**), as well as the end-to-end time for CORTS (**$E2E_{CORTS}$**), C2RTS (**$E2E_{C2RTS}$**), and STARTS (**$E2E_{STARTS}$**). Table 7 also presents **TESTAll**, which is a strategy that just runs all test cases without performing any RTS analysis. We use the **TESTAll** strategy time as the baseline and compared the end-to-end times of the RTS approaches with it. Table 7 displays, per subject, the overall cumulative time for all the 100 revisions of the subject.

It is important to mention that for static class-level RTS, we did not separately measure the *selection phase* time (i.e., **$SELECT_{STARTS\text{-}like}$** time in Table 7) using STARTS. Instead, we developed a STARTS-like tool that functions similarly to STARTS by using `jdeps` to extract class dependencies and building a class-level dependency graph. However, the STARTS-like tool utilizes `JGraphT` for graph construction and analysis, whereas STARTS uses the custom, faster yasgl library [34]. To ensure a fair comparison, we compared the *selection phase* time of CORTS and C2RTS with STARTS-like, since all three use `JGraphT` for graph operations. Additionally, STARTS does not provide specific commands to report the exact *selection phase* time separately from

Table 7
Selection phase time and end-to-end testing time in seconds.

Subject	$SELECT_{CORTS}$	$SELECT_{C2RTS}$	$SELECT_{STARTS\text{-}like}$	TESTAll	$E2E_{CORTS}$	$E2E_{C2RTS}$	$E2E_{STARTS}$
commons-math	**4.153**	11.859	23.209	11,042.579	9572.399	6023.054	**3664.108**
commons-configuration	**1.445**	6.283	10.329	2579.310	2132.351	1826.158	**1592.784**
commons-compress	**0.871**	1.819	4.263	819.969	730.118	621.127	**572.821**
commons-collections	**1.206**	2.586	8.957	1287.695	809.505	259.606	**124.392**
commons-dbcp	**0.273**	0.526	1.237	8252.194	7466.198	5032.529	**4653.431**
commons-io	**0.436**	1.119	2.275	4975.982	3106.881	2123.687	**1828.807**
commons-lang	**0.852**	1.583	3.781	1621.582	1131.397	1067.314	**776.306**
commons-validator	**0.251**	0.569	0.908	168.435	117.999	82.979	**38.188**
commons-pool	**0.252**	0.456	0.809	31,183.825	26,125.138	26,026.082	**24,691.189**
JFreeChart	**2.235**	10.406	38.091	538.722	305.523	274.989	**149.916**
jankotek.mapdb	**1.195**	9.279	13.849	56,929.985	54,353.748	40,567.913	**38,828.116**
OpenTripPlanner	**8.167**	20.198	45.879	17,672.936	17,318.823	17,330.737	**14,778.681**

$SELECT_X$ is the summation (per subject) of the overall execution time taken by an RTS approach ($_X$) for the test selection process (i.e., construct and analyze dependency graph to select tests).
$E2E_X$ is the summation (per subject) of the overall end-to-end execution time taken by an RTS approach ($_X$).
TESTAll is the summation (per subject) of the overall time taken to just run all test cases.

other RTS phases and operations, e.g., computing and storing smart checksums. For the end-to-end time, we reported the time taken by STARTS (i.e., **$E2E_{STARTS}$** time in Table 7) instead of the STARTS-like tool.

The CORTS had the shortest *selection phase* time across all the 12 subjects, followed by C2RTS, with STARTS-like taking the longest. For instance, in the `JFreeChart` project, CORTS took 2.235 s, C2RTS took 10.4 s, and class-level RTS took 38.09 s. By averaging the *selection phase* time across the 12 subjects, we found that the overall average *selection phase* time was 1.77 s for CORTS, 5.56 s for C2RTS, and 12.79 s for STARTS. This is because the dependency graphs for CORTS and C2RTS are smaller compared to the static class-level dependency graph. The reported *selection phase* times suggest that CORTS and C2RTS scale better for larger graphs, requiring less time to construct and analyze dependency graphs for test case selection.

All three RTS approaches, i.e., CORTS, C2RTS, and STARTS, reduced the overall end-to-end testing time compared to the **TESTAll** baseline across all 12 subjects. For example, in the `commons-pool` project, which comes with long running JUnit test cases, the **TESTAll** took 31,183 s for running all test cases of the subject, which is the total time summed-up for all the 100 revisions of this subjects, while the overall end-to-end testing time was 26,125 s for CORTS, 26,026 s for C2RTS, and 24,691 for STARTS. For all the subjects, STARTS had the shortest end-to-end time due to its highest reduction in the test suite size, followed by C2RTS, while CORTS took the longest time. By averaging the end-to-end testing time across the 12 subjects, we found that the overall average time was 11,422.76 s for **TESTALL**, 10,264.17 s for CORTS, 8436.34 s for C2RTS, and 7641.56 s for STARTS. Despite that CORTS had the longest time, it still showed reduction in the end-to-end testing time compared to the **TESTAll** strategy, indicating its practical value in regression testing. C2RTS achieved better results than CORTS in reducing the end-to-end time, showing that such a hybrid RTS technique can provide balancing between component- and class-level, where it achieves reduction in the regression testing time while still maintaining high safety and scalability, making it suitable for large JPMS-based programs where balance between performance and safety is critical.

4.8. Results discussion

Balancing metrics across RTS approaches. The evaluation results highlight a trade-off between key regression test selection (RTS) metrics: safety violation, precision violation, test suite reduction, and end-to-end testing time. Specifically, STARTS outperforms CORTS and C2RTS in terms of precision violation, test suite reduction, and test execution time, while CORTS and C2RTS achieve lower safety violation rates.

CORTS and C2RTS emphasize safety by employing a coarser granularity (component-level) in dependency analysis. This strategy ensures that fewer impacted test cases are missed, reducing safety violations. However, this comes at the expense of increased test suite size and higher precision violations, as non-impacted test cases may also be selected. On the other hand, STARTS operates at the class level, and thus, achieves higher test suite reduction and precision, minimizing the selection of unnecessary test cases. However, this finer granularity can lead to higher safety violations compared to component-level RTS. This is because static class-level RTS may miss dynamic dependencies related to reflection and dynamic class loading. In contrast, CORTS and C2RTS can account for such dependencies as they are explicitly declared in the `module-info.java` files.

RTS execution time. The dependency graph construction and analysis time for CORTS and C2RTS is significantly shorter than for STARTS. This improvement is due to the smaller size of component-level dependency graphs. However, the time spent on dependency graph processing constitutes a minor fraction of the overall end-to-end testing time, which is dominated by test execution. Consequently, STARTS significantly outperformed CORTS and C2RTS in terms of the overall end-to-end testing time, as it obtained higher reduction in test suite size and smaller precision violations.

While dependency graph efficiency does not drastically impact total end-to-end testing time, it plays a crucial role in continuous integration (CI) environments by enabling faster feedback cycles for developers. Rapid test selection allows for immediate identification of impacted tests, reducing delays in iterative development workflows.

Scenarios for Class- versus Component-level RTS. The choice of RTS approach depends on the application's context and requirements. For example, class-level RTS is preferable in resource-constrained environments where test execution cost and time are critical, e.g., mobile app development pipelines. It is also preferable for applications with frequent but small changes where the likelihood of missing impacted test cases is minimal, such as utility libraries or microservices with isolated functionality. On the other hand, component-level RTS (e.g., CORTS and C2RTS) can be more preferable in safety-critical domains where ensuring comprehensive test coverage outweighs reducing regression testing execution time, such as in component-based adaptive systems with fault tolerance mechanisms [35], aircraft [36], aerospace or other safety-critical systems [37,38]. Additionally, component-level RTS can be more appropriate for large-scale, monolithic enterprise systems with complex interdependencies across components, and we plan to investigate this direction in the future.

5. Threads to validity

External validity. External validity affects the generalizability of our results. One external validity threat is the use of 12 Java projects which might not be representative, so our results may not generalize. However, the selected subjects are widely used to evaluate RTS approaches [12,17–19,39], vary in size, application domain, and number

of test classes, which reduces this threat. Additionally, the results could differ for larger Enterprise Resource Planning (ERP) systems, but we anticipate that component-level RTS would be even more scalable and reusable in such cases compared to class-level RTS approaches. We plan to investigate this direction in the future.

Another threat to the external validity of our experimental results is the use of OO2CB tool which uses the ACDC tool to determine the components of OO Java projects. Therefore, OO2CB inherits the shortcoming of the ACDC in determining components of OO Java projects. Using other component recovery tools, such as Architecture Recovery using Concerns (ARC) [31], Bunch [32], or Weighted Combined Algorithm (WCA) [40], could change the RTS results. To reduce this threat, we leveraged the best-performing component recovery tool, i.e., ACDC [33], as concluded in prior comparative studies of recovery techniques [30].

Internal validity. An internal factor that can affect the outcome is the possible errors in the implementations of CORTS and C2RTS. To mitigate this threat, we built our implementation on mature tools (i.e., JGraphT [41] and jdeps [27]) and tested it thoroughly.

Another threat to internal validity is the use of the OO2CB tool [22] to generate the (JPMS) component-based projects from the subjects according to the least-privilege security principle. The threat is related to the false positives caused by the static analysis of class dependencies. Static analysis results may overestimate the communications between components which might lead to granting a component more privileges than it needs in the resulted CB app. Consequently, this could impact the results of our experiments, especially the precision and reduction in test suite size. However, OO2CB uses the BCEL tool [42], a widely-used library in the industry to analyze Java apps, which mitigates this threat. Furthermore, OO2CB defines all port types in the component-based applications except the ones responsible for Java reflection and dynamic class loading techniques, i.e., the `open` and `opens with` ports. This limitation impacted the safety violation results yielded by CORTS and C2RTS. We expect that these results could be even better if CORTS and C2RTS are applied to component-based applications that utilize the `opens with` port to denote dependencies deriving from reflection and dynamic class loading among components. This area will be a focus of our future research endeavors.

Construct validity. In general, we could have used other metrics (e.g., test coverage and fault detection ability) to evaluate the effectiveness of CORTS and C2RTS. However, we used the most common metrics in the research literature: safety violation, precision violation, reduction in test suite size, and reduction in end-to-end test execution time. We also used the reduction in dependency graph size as measure to evaluate the scalability of CORTS and C2RTS to large subjects.

Another threat to construct validity is that we chose Ekstazi as the ground truth against which to evaluate the static RTS techniques, e.g., we computed the safety and precision violations with respect to Ekstazi. Although Ekstazi is a state-of-the-art, and is recognized as a leading and accessible dynamic class-level RTS tool, it might not encompass the entirety of benchmarks for all RTS scenarios.

Conclusion Validity. We only used 12 subjects to evaluate CORTS and C2RTS. The use of additional subjects could affect the conclusions of the evaluation. To reduce this threat, we used large real-world Java projects that have been used in other experiments for fair comparison.

6. Related work

RTS can reduce regression testing efforts and has been studied for over three decades [14,15]. Below we summarize the existing dynamic and static RTS approaches.

Dynamic RTS. Many graph-walk approaches address the problem of RTS. Rothermel and Harrold [2] propose a safe approach for RTS for procedural programs. The algorithm uses control-flow graphs (CFG) to represent each procedure in a program *P* and its modified version *P'*. Each node in a CFG represents a simple or conditional statement, and each edge represents the flow of control between statements. Entities affected by modifications are selected by traversing in parallel the CFGs of *P* and *P'*, and when the target entities of like-labeled CFG edges in *P* and *P'* differ, the edge is added to the set of affected entities. Thereafter, Rothermel and Harrold extended the CFG-based algorithm for C++ using Inter-procedural Control-Flow Graphs (ICFG) [43]. Harrold et al. [44] further extended the CFG approach for Java software using the Java Inter-class Graph (JIG) to handle Java features and incomplete programs.

Vokolos and Frankl [45] consider RTS based on text differencing using the Unix `diff` command. The approach compares the original code version with the modified version to identify the modified statements, and selects test cases that exercise code blocks containing these statements.

To improve the efficiency of dynamic RTS, a number of techniques at coarser granularity (e.g., method- or class-level) rather than the finer granularity CFG level (e.g., statement-level) were proposed. Ren et al. [46] and Zhang et al. [47] applied change-impact analysis at the method-level, based on call graphs techniques, to improve RTS. Recent RTS approaches [12,18,19] were proposed to make RTS more cost-effective in modern software systems by focusing on class-level RTS that (1) identifies changes at the class level and (2) computes dependencies from test cases to the classes under test. Additionally, these approaches consider a test class as a test case, and thus, select test classes instead of test methods. Gligoric et al. [12] proposed Ekstazi, an approach that tracks dynamic dependencies of test cases at the class level and selects test cases that traverse modified classes. Ekstazi is a safe RTS approach, and its safety is based on the formally proven safety of the change-based RTS approach [48]. Zhang [19] proposed HyRTS, which is a dynamic and hybrid approach that supports analyzing the adapted classes at multiple granularity levels (i.e., method and class levels) to improve the precision and selection time. Running HyRTS using the class-level mode produces the same RTS results as Ekstazi [19]. Thus, HyRTS was not considered in our experimental evaluation.

While these dynamic RTS techniques can be safe, they require dynamic test coverage information which may be absent, costly to collect, or require prohibitive instrumentation (e.g., for non-deterministic or real-time code). On the other hand, our proposed approaches, CORTS and C2RTS are static, but still can capture runtime information represented in the module descriptors (i.e., module-info.java file) such as dependencies related to dynamic class loading and reflection represented using the `opens with` directive.

Static RTS. Kung et al. [49,50], Hsia et al. [51], and White and Abdullah [25] proposed firewall-based approaches. The firewall contains the changed classes and their dependent classes, where the dependent classes are identified based on static analysis. Test cases that traverse classes in the firewall are selected. Jang et al. [52] apply firewall-based RTS at the method level to C++ software. They identify firewalls around all the methods affected by a change and select all the test cases exercising these methods for regression testing. Ryder and Tip [53] proposed a call-graph-based static change-impact analysis technique and evaluated only one call-graph analysis on 20 revisions of one project [54]. Skoglund and Runeson's [48] proposed a change-based approach that only selects those test cases that exercise the changed classes. ChEOPSJ [55,56] is a static change-based approach that uses the FAMIX model to represent software entities including test cases and building dependencies between them. These approaches use fine-grained information such as constructor calls and method invocation statements to build dependencies between software entities.

Legunsen et al. [17,18] proposed STARTS, which is a static RTS approach that is based on the idea of the class-level firewall. STARTS builds a dependency graph of program types based on compile-time information, and selects test cases that can reach changed types in the transitive closure of the dependency graph. Yu et al. [16] evaluated

method-level and class-level static RTS in continuous integration environments. Class-level RTS was determined to be more practical and time-saving than method-level RTS.

Gyori et al. [20] compared variants of dynamic and static class-level RTS with project-level RTS in the Maven Central open source ecosystem. An ecosystem may contain a large number interconnected projects, where client projects transitively depend on library projects. Project-level RTS identifies changes at the project level and computes dependencies from test cases to projects. When a library changes, then all test cases in the library and all test cases in all the library's transitive clients are selected. Class-level RTS was found to be less costly than project-level RTS in terms of reduction in test suite size.

Shi et al. [8] focused on optimizing RTS in continuous integration (CI) environments. They compared module- and class-level RTS techniques in the Travis cloud-based CI environment, and developed a hybrid RTS technique, called GIBstazi, that combines aspects of the module- and class-level RTS techniques. Their work focuses on Maven modules (i.e., build-system modules) utilizing techniques like the Git Inferred Build (GIB) to optimize test selection based on module dependencies determined by the build system (i.e., focusing on build-time dependencies). While the work of Shi et al. [8] is more aligned with multi-module Java applications structured with the Maven build system, our approaches, CORTS and C2RTS, utilize JPMS modules that emphasize dependencies -including runtime dependencies specified using the `opens with` directive- and encapsulation according to the least privilege concept. Although we did not empirically compare the precision of JPMS-module level RTS to build-system module level RTS, we anticipate that the latter may be less precise in detecting affected tests due to the broader scope of build-time dependencies. On the other hand, JPMS-based RTS could potentially offer more precise and safer test selection due to the explicit module dependencies and encapsulation provided by JPMS. As a future work, we plan to transform multi-module maven-based Java applications into their JPMS-utilizing counterparts to further evaluate the efficacy of CORTS and C2RTS in such environments.

Overall, CORTS and C2RTS are similar to the described static RTS approaches in terms of applying the firewall impact analysis technique, but at the module-level rather than the class- and method-levels. However, unlike the existing static RTS techniques, our proposed approaches can capture runtime information that are explicitly included in the module descriptor files.

7. Conclusions and future work

As software systems become increasingly complex and large, especially with the implementation of the Java Platform Module System (JPMS), traditional regression test selection (RTS) techniques at the method and class levels often face challenges in efficiency and resource management. This research was driven by the desire to refine RTS for Java applications modularized with JPMS. This research leverages component-level granularity and provides a substantial foundation for advancing RTS practices tailored to modern Java applications, presenting a strong case for the adoption of component-level analysis in professional and large-scale development environments.

We introduced two novel static component-based RTS approaches, CORTS and its variant C2RTS, tailored for component-based Java software systems modularized with JPMS. CORTS constructs a module-level dependency graph using architectural metadata from module descriptor files to determine the impact of changes and select relevant test cases. C2RTS extends this by incorporating class-level analysis for modified modules, offering a hybrid approach that balances granularity to improve precision while maintaining safety. Our evaluation of CORTS and C2RTS on real-world software systems demonstrated improvements, in terms of safety, over static class-level RTS paradigms. Additionally, both CORTS and C2RTS reduced the dependency graph size compared to static class-level RTS, thus, providing an evidence that component-level RTS can be more scalable for large-scale projects. Additionally, C2RTS demonstrated better precision than CORTS, thus balancing between safety and precision while still reducing the size of the static dependency graph compared to static class-level RTS. Both, CORTS and C2RTS, reduced the end-to-end testing time in comparison to running all test cases without performing RTS.

We plan to extend the application of CORTS and C2RTS to large-scale enterprise Java systems. Furthermore, it is critical to acknowledge that the component recovery tools used in our experimental evaluations, such as ACDC, do not possess the capability to detect dynamic dependencies, such as those involving reflection, and consequently, do not incorporate these dependencies into the module descriptor files. Moving forward, we plan to explore and experiment with alternative component recovery tools that can better capture dynamic dependencies and reflection. Additionally, we plan to explore the application of our RTS approaches in the context of Java 9 modular applications when treating the modularized Java Runtime Environment (JRE) and third-party libraries, along with their dependencies, as part of the CB application. This investigation aims to evaluate the impact of reduced runtime size, e.g., only including the required modules of the JRE and third-party libraries, on the RTS performance.

CRediT authorship contribution statement

Mohammed Al-Refai: Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision, Software, Resources, Project administration, Methodology, Investigation, Formal analysis, Data curation, Conceptualization. **Mahmoud M. Hammad:** Writing – review & editing, Visualization, Validation, Resources, Formal analysis.

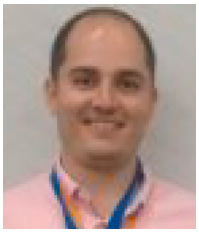
Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

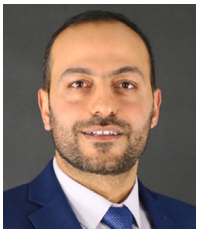
References

[1] A. Bertolino, Software testing research: Achievements, challenges, dreams, in: 2007 Future of Software Engineering, IEEE Computer Society, 2007 pp. 85–103.
[2] G. Rothermel, M.J. Harrold, A safe, efficient regression test selection technique, ACM Trans. Softw. Eng. Methodol. 6 (2) (1997) 173–210.
[3] M.J. Harrold, Testing evolving software, J. Syst. Softw. 47 (2–3) (1999) 173–181.
[4] H.K.N. Leung, L.J. White, Insights into regression testing, in: Proceedings of Conference on Software Maintenance, IEEE, Miami, FL, USA, 1989, pp. 60–69.
[5] P.K. Chittimalli, M.J. Harrold, Recomputing coverage information to assist regression testing, IEEE Trans. Softw. Eng. 35 (4) (2009) 452–469.
[6] E. Engström, P. Runeson, A qualitative survey of regression testing practices, in: International Conference on Product Focused Software Process Improvement, Springer, 2010, pp. 3–16.
[7] R. Greca, B. Miranda, A. Bertolino, State of practical applicability of regression testing research: A live systematic literature review, ACM Comput. Surv. 55 (13s) (2023) 1–36.
[8] A. Shi, P. Zhao, D. Marinov, Understanding and improving regression test selection in continuous integration, in: 2019 IEEE 30th International Symposium on Software Reliability Engineering, ISSRE, IEEE, 2019, pp. 228–238.
[9] W. Sun, X. Xue, Y. Lu, J. Zhao, M. Sun, Hashc: Making deep learning coverage testing finer and faster, J. Syst. Archit. 144 (2023) 102999.
[10] Y. Lu, K. Shao, J. Zhao, W. Sun, M. Sun, Mutation testing of unsupervised learning systems, J. Syst. Archit. 146 (2024) 103050.
[11] Testing at the speed and scale of Google, 2011, http://google-engtools.blogspot.com/2011/06/testing-at-speed-and-scale-of-google.html.
[12] M. Gligoric, L. Eloussi, D. Marinov, Practical regression test selection with dynamic file dependencies, in: Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA'15, ACM, Baltimore, MD, USA, 2015, pp. 211–222.
[13] L.C. Briand, Y. Labiche, S. He, Automating regression test selection based on UML designs, J. Inf. Softw. Technol. 51 (1) (2009) 16–30.

[14] E. Engström, P. Runeson, M. Skoglund, A systematic review on regression test selection techniques, Inf. Softw. Technol. 52 (1) (2010) 14–30.
[15] S. Yoo, M. Harman, Regression testing minimization, selection and prioritization: A survey, J. Softw. Test. Verif. Reliab. 22 (2) (2012) 67–120.
[16] T. Yu, T. Wang, A study of regression test selection in continuous integration environments, in: S. Ghosh, R. Natella (Eds.), Proceedings of the 29th International Symposium on Software Reliability Engineering, ISSRE'18, IEEE, Memphis, TN, USA, 2018, pp. 135–143.
[17] O. Legunsen, F. Hariri, A. Shi, Y. Lu, L. Zhang, D. Marinov, An extensive study of static regression test selection in modern software evolution, in: J. Cleland-Huang, Z. Su (Eds.), Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE'16, ACM, Seattle, WA, USA, 2016, pp. 583–594.
[18] O. Legunsen, A. Shi, D. Marinov, STARTS: Static regression test selection, in: M. Di Penta, T.N. Nguyen (Eds.), Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE'17, IEEE Press, Urbana-Champaign, IL, USA, 2017, pp. 949–954.
[19] L. Zhang, Hybrid regression test selection, in: M. Chechik, M. Harman (Eds.), Proceedings of the 40th International Conference on Software Engineering, ICSE'18, IEEE, Gotheburg, Sweden, 2018, pp. 199–209.
[20] A. Gyori, O. Legunsen, F. Hariri, D. Marinov, Evaluating regression test selection opportunities in a very large open-source ecosystem, in: S. Ghosh, R. Natella (Eds.), Proceedings of the 29th International Symposium on Software Reliability Engineering, ISSRE'18, IEEE, Memphis, TN, USA, 2018, pp. 112–122.
[21] JPMS. http://openjdk.java.net/projects/jigsaw/spec/.
[22] M.M. Hammad, I. Abueisa, S. Malek, Tool-assisted componentization of Java applications, in: 2022 IEEE 19th International Conference on Software Architecture, ICSA, 2022, pp. 36–46, http://dx.doi.org/10.1109/ICSA53651.2022.00012.
[23] OpenJDK: Jigsaw project. https://openjdk.java.net/projects/jigsaw/.
[24] R.N. Taylor, N. Medvidovic, E.M. Dashofy, Software architecture: foundations, theory, and practice, Google Sch. Google Sch. Digit. Libr. Digit. Libr. (2009) (2009).
[25] L.J. White, K. Abdullah, A firewall approach for regression testing of object-oriented software, in: Proceedings of the 10th International Software Quality Week, QW'97, San Francisco, CA, USA, 1997.
[26] D. Michail, J. Kinable, B. Naveh, J.V. Sichi, JGraphT—A Java library for graph data structures and algorithms, ACM Trans. Math. Software 46 (2) (2020).
[27] jdeps: The Java class dependency analyzer. Available from Oracle: https://docs.oracle.com/javase/8/docs/technotes/tools/unix/jdeps.html.
[28] A. Shi, A. Gyori, M. Gligoric, A. Zaytsev, D. Marinov, Balancing trade-offs in test-suite reduction, in: A. Orso, M.-A. Storey (Eds.), Proceedings of the 22nd International Symposium on Foundations of Software Engineering, FSE'14, ACM, Hong Kong, China, 2014, pp. 246–256.
[29] 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, 2019, pp. 560–571, http://dx.doi.org/10.1109/ICSE.2019.00067.
[30] J. Garcia, I. Ivkovic, N. Medvidovic, A comparative analysis of software architecture recovery techniques, in: 2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE, IEEE, 2013, pp. 486–496.
[31] J. Garcia, D. Popescu, C. Mattmann, N. Medvidovic, Y. Cai, Enhancing architectural recovery using concerns, in: 2011 26th IEEE/ACM International Conference on Automated Software Engineering, ASE 2011, IEEE, 2011, pp. 552–555.
[32] B.S. Mitchell, S. Mancoridis, On the automatic modularization of software systems using the bunch tool, IEEE Trans. Softw. Eng. 32 (3) (2006) 193–208.
[33] V. Tzerpos, R.C. Holt, ACDC: an algorithm for comprehension-driven clustering, in: Proceedings Seventh Working Conference on Reverse Engineering, IEEE, 2000, pp. 258–267.
[34] Yet another simple graph library. https://github.com/TestingResearchIllinois/yasgl.
[35] M. Stoicescu, J.-C. Fabre, M. Roy, Architecting resilient computing systems: A component-based approach for adaptive fault tolerance, J. Syst. Archit. 73 (2017) 6–16.
[36] H. Usach, J.A. Vila, C. Torens, F. Adolf, Architectural design of a safe mission manager for unmanned aircraft systems, J. Syst. Archit. 90 (2018) 94–108.
[37] Z. Yang, Z. Qiu, Y. Zhou, Z. Huang, J.-P. Bodeveix, M. Filali, C2AADL_Reverse: A model-driven reverse engineering approach to development and verification of safety-critical software, J. Syst. Archit. 118 (2021) 102202.
[38] I. Allende, N. Mc Guire, J. Perez, L.G. Monsalve, R. Obermaisser, Towards Linux based safety systems—A statistical approach for software execution path coverage, J. Syst. Archit. 116 (2021) 102047.
[39] M.K. Shin, S. Ghosh, L.R. Vijayasarathy, An empirical comparison of four Java-based regression test selection techniques, J. Syst. Softw. 186 (2022) 111174, http://dx.doi.org/10.1016/j.jss.2021.111174, URL https://www.sciencedirect.com/science/article/pii/S0164121221002582.
[40] O. Maqbool, H. Babri, Hierarchical clustering for software architecture recovery, IEEE Trans. Softw. Eng. 33 (11) (2007) 759–780.
[41] B. Naveh, J.V. Sichi, JGraphT a free Java graph library, 2011.
[42] BCEL documentation available at http://jakarta.apache.org/bcel/.
[43] G. Rothermel, M.J. Harrold, J. Dedhia, Regression test selection for C++ software, Softw. Test. Verif. Reliab. 10 (2) (2000) 77–109.
[44] M.J. Harrold, J.A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S.A. Spoon, A. Gujarathi, Regression test selection for Java software, in: J. Vlissides (Ed.), Proceedings of the 16th Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA'01, ACM, Tampa, FL, USA, 2001, pp. 312–326.
[45] F. Vokolos, P.G. Frankl, Empirical evaluation of the textual differencing regression testing technique, in: Proceedings of the International Conference on Software Maintenance, SM'98, Bethesda, MD, USA, 1998, pp. 44–53.
[46] X. Ren, F. Shah, F. Tip, B.G. Ryder, O. Chesley, Chianti: a tool for change impact analysis of java programs, in: Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, 2004, pp. 432–448.
[47] L. Zhang, M. Kim, S. Khurshid, Faulttracer: a change impact and regression fault analysis tool for evolving java programs, in: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, 2012, pp. 1–4.
[48] M. Skoglund, P. Runeson, Improving class firewall regression test selection by removing the class firewall, Int. J. Softw. Eng. Knowl. Eng. 17 (3) (2007) 359–378.
[49] D.C. Kung, J. Gao, P. Hsia, J. Lin, Y. Toyoshima, Class firewall, test order, and regression testing of object-oriented programs, J. Occup. Organ. Psychol. 8 (2) (1995) 51–65.
[50] D.C. Kung, J. Gao, P. Hsia, Y. Toyoshima, C. Chen, On regression testing of object-oriented programs, J. Syst. Softw. 32 (1) (1996) 21–40.
[51] P. Hsia, X. Li, D.C.-H. Kung, C.-T. Hsu, L. Li, Y. Toyoshima, C. Chen, A technique for the selective revalidation of OO software, J. Software: Evol. Process. 9 (4) (1997) 217–233.
[52] Y.K. Jang, M. Munro, Y.R. Kwon, An improved method of selecting regression tests for C++ programs, J. Softw. Maint. Evol. 13 (5) (2011) 331–350.
[53] B.G. Ryder, F. Tip, Change impact analysis for object-oriented programs, in: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, 2001, pp. 46–53.
[54] X. Ren, F. Shah, F. Tip, B.G. Ryder, O. Chesley, J. Dolby, Chianti: A Prototype Change Impact Analysis Tool for Java, Tech. Rep., Rutgers University, 2003.
[55] Q.D. Soetens, S. Demeyer, A. Zaidman, Change-based test selection in the presence of developer tests, in: A. Cleve, F. Ricca (Eds.), Proceedings of the 17th European Conference on Software Maintenance and Reengineering, CSMR'13, IEEE, Genoa, Italy, 2013, pp. 101–110.
[56] Q.D. Soetens, S. Demeyer, A. Zaidman, J. Pérez, Change-based test selection: An empirical evaluation, Empir. Softw. Eng. (2015) 1–43.



Dr. Mohammed Al-Refai is an Assistant Professor in the Computer Science Department within the Computer and Information Technology School at the Jordan University of Science and Technology (JUST). Al-Refai's research focuses on various areas within software engineering, including model-driven development, model-based testing, software architecture, software testing, regression test selection and prioritization, software security, and the integration of fuzzy logic and machine learning in software engineering applications. Al-Refai earned his Ph.D. in Computer Science from Colorado State University, Fort Collins, Colorado, under the supervision of Prof. Sudipto Ghosh. He also holds M.S. and B.S. in Computer Science from Jordan University of Science and Technology. Al-Refai is a member of the Association for Computing Machinery (ACM) and the Institute of Electrical and Electronics Engineers (IEEE).



Dr. Mahmoud Hammad is an Associate Professor in the Software Engineering Department within the Computer and Information Technology School at the Jordan University of Science and Technology (JUST). He is also the director of the Center for E-Learning and Open Educational Resources. Hammad's research interests are in the field of software engineering, specifically in the area of software architecture, self-adaptive software systems, mobile computing, software analysis, software security, natural language processing and machine learning. Hammad received his Ph.D. in Software Engineering from the University of California, Irvine (UCI)

under the supervision of Prof. Sam Malek . During his Ph.D., Hammad developed a self-protecting Android software system , an Android software system that can monitor itself and adapt (change) its behavior at runtime to keep the system secure and protected from Inter-Component Communication attacks at all times. Hammad received his M.S. in Software Engineering from George Mason University, VA, USA and B.S. in Computer Science from Yarmouk University, Jordan . Hammad is a member of the Association of Computing Machinery (ACM), ACM Special Interest Group on Software Engineering (SIGSOFT), and the Institute of Electrical and Electronics Engineers (IEEE). https://hammadmahmoud.github.io/