DIRTS: Dependency Injection Aware Regression Test Selection

Simon Hundsdorfer[*]
Technical University of Munich
Munich, Germany

Daniel Elsner[*]
Technical University of Munich
Munich, Germany

Alexander Pretschner
Technical University of Munich
Munich, Germany

Abstract*—Regression test selection (RTS) aims to reduce regression testing effort by selecting only those tests that are affected by introduced changes. RTS techniques are considered to be *safe* if they select all affected test cases. Several supposedly safe RTS tools have been developed over the past decades, lately especially for Java projects. However, recent studies have shown that state-of-the-art RTS tools for Java can become unsafe when confronted with dependency injection (DI) mechanisms: despite the widespread use of DI frameworks in Java projects, no existing technique acknowledges DI-related changes. In this paper, we analyze the reasons behind unsafe RTS behavior for DI-related changes and develop DIRTS, a novel DI-aware RTS tool for Java. To counteract effects of DI on RTS, DIRTS efficiently analyzes source code annotations and metadata employed by popular DI frameworks, and generates a dependency graph including edges for dynamically injected objects. We evaluate DIRTS on 228 commits from 9 open-source Java projects that use DI. Our results indicate that in 33.3% of those commits DI-related changes affect some tests, and in 3.1% (7) DIRTS identifies affected tests that are clearly missed by the static RTS tool STARTS. Still, DIRTS is comparatively efficient and precise. We publish DIRTS[1,2] as an RTS tool that can either be used as a safety extension for existing RTS tools or as a standalone RTS solution.

***Index Terms*—Software testing, regression test selection, dependency injection, static program analysis, cross-language links**

I. INTRODUCTION

Regression testing is regularly performed on software systems to ensure that changes have not inadvertently affected existing system behavior. The simplest yet expensive *retest-all* strategy is to execute every test case in the test suite after each introduced change. Yet, with increasingly large test suites and shorter development lifecycles this strategy often becomes infeasible [2]. Regression test selection (RTS) aims to minimize the regression testing effort by only re-executing tests that may yield a different result due to changes in the code [3]–[5].

An RTS technique is *safe*, if all affected tests are correctly identified [6]. To relate changes to tests they affect, RTS techniques typically maintain a dependency graph of code entities, *e.g.,* functions, for each test. Then, if a code entity has changed, all tests that depend on the entity are selected. Based on the type of program analysis they apply to generate the dependency graph, RTS techniques can further be divided into *static* and *dynamic* techniques [7].

Several static and dynamic RTS techniques have been proposed, specifically for languages targeting the Java virtual machine (JVM), such as Java [7]–[16]. Most of these techniques were originally designed to be safe under the assumption of Java code changes. However, Zhu *et al.* [17] discovered that this assumption may not hold true in practice: using their tool RTSCHECK, they found safety violations in state-of-the-art RTS tools due to changes to non-Java (source) files (*e.g.,* XML files). More recently, Shi *et al.* [18] as well as Elsner *et al.* [12] identified cases where RTS tools are not able to recognize Java or XML code changes which may alter the run-time behavior of tests through dependency injection (DI), where objects are dynamically created and injected. Although DI is a widely adopted design principle to reduce coupling in object-oriented software—it is heavily used in Java EE and Spring [19], [20]—contemporary RTS tools lack adequate DI support.

In this paper, we investigate why RTS is prone to be unsafe when confronted with DI mechanisms. Based on our findings, we design and implement DIRTS, a static RTS tool for Java, which aims to address the shortcomings of existing RTS tools regarding DI. DIRTS leverages efficient static Java source code analysis to construct a dependency graph with additional edges related to DI. Since DI frameworks often allow configuration through XML, DIRTS also acknowledges these cross-language links and parses metadata from XML files [21]. Currently, DIRTS supports three of the most popular DI frameworks in Java: Spring[3], Guice[4], and CDI[5]. To give software engineers flexibility and stick to established development processes and tooling, DIRTS can either be used as a safety extension for existing RTS tools or as a standalone RTS tool.

We evaluate DIRTS in an empirical study on 228 commits from 9 open-source Java projects that make use of DI. To find commits where existing RTS tools might be unsafe, we compare DIRTS with STARTS, a static RTS tool from prior research [7], [8]. We find that in 33.3% of the analyzed commits DI-related

[*]Both authors contributed equally

[1]DIRTS on GitHub: https://github.com/tum-i4/dirts

[2]DIRTS demo video: [1]

[3]Spring: https://spring.io/

[4]Guice: https://github.com/google/guice

[5]Jakarta CDI: https://jakarta.ee/specifications/cdi

changes affect some tests, whereas STARTS misses to select affected tests in 3.1% (7) commits. Despite its superior safety for DI-related changes, DIRTS has comparable efficiency and precision and, contrary to STARTS, does not require a fully compiled Java workspace, giving more flexibility for selective build optimization [12].

In summary, this paper makes the following contributions:

- *DI-aware RTS:* We are the first to investigate the problem of unsafe RTS behavior related to DI mechanisms.
- DIRTS *tool:* We implement DIRTS, a static RTS tool for Java that can be used as a safety extension for existing RTS tools as well as a standalone RTS solution, where it supports method- and class-level test selection.
- *Empirical study:* We present an empirical study to evaluate the efficiency and effectiveness of DIRTS and the prevalence of DI-related RTS safety violations in 9 open-source Java projects. Not only is DIRTS safer than the RTS tool STARTS, but also comparatively efficient and effective.

II. BACKGROUND & MOTIVATING EXAMPLES

A. Dependency Injection

Dependency injection (DI) is an object-oriented design technique used to reduce coupling in modern software systems [22]. As pointed out by Saeed [23], it is basically inevitable to have certain components depend on others in software development. Using DI, components only need to declare where to inject components they depend on (*i.e.,* dependencies) and the instantiation of the corresponding objects is done by a framework [23]. These injected objects are often referred to as *beans* [24]. To identify the implementation (*e.g.,* a Java class) that is supposed to be instantiated and injected, mappings between developer-defined keys and associated implementations are typically either defined through XML metadata or source code annotations [25].

B. DI-related Safety Violations in Regression Test Selection

In the context of RTS, Zhu *et al.* [17] identified safety violations of RTS tools such as STARTS or EKSTAZI due to their unawareness of non-Java (source) files. Since cross-language links to XML are often employed to configure DI frameworks [21], projects using DI are particularly prone to these violations. Shi *et al.* [18] and Elsner *et al.* [12] further report (potential) safety violations related to missing edges in static and dynamic dependency graphs, respectively, which are introduced through DI frameworks' source code annotations. When for example the injection priority of beans is changed, this may affect the run-time behavior of a test without modifying any of the entities covered by the test before the change [12]. For the rest of the paper, we collectively refer to changes related to DI mechanics as *DI-related changes*.

C. Illustrative Examples

To motivate and understand how DI-related changes can lead to RTS safety problems, we discuss two illustrative source code examples in the following. Both examples can result in safety violations for the well-known Java RTS tools STARTS [7], [8], EKSTAZI [9], [10], and HYRTS [15].

In both examples, the DI frameworks aim to inject an object of an implementation of the interface `DataSource` shown in Listing 1 (loosely inspired by `javax.sql.DataSource`).

Listing 1: Interface used in illustrative examples

```
interface DataSource {
   Connection getConnection();
}
```

1) Safety Violations due to XML Configuration: The first example takes into account that static and dynamic RTS tools are typically not designed to analyze cross-language links to XML files, which may contain relevant metadata for DI frameworks [12], [17], [26]. For the purpose of demonstration, we chose CDI, the DI framework integrated into the Jakarta EE (formerly Java EE) platform, though, similar scenarios can be constructed for other DI frameworks such as Spring.

Before the changes are introduced, there is an implementation of `DataSource` named `InMemoryDB` shown in Listing 2, which is injected across the application using CDI.

Listing 2: Implementation of `DataSource` before change

```
@Alternative
class InMemoryDB implements DataSource {
   @Override
   public Connection getConnection() {
      return InMemoryDBFactory.create();
   }
}
```

Now suppose, a new implementation of `DataSource`, `PostgresDB` is added (see Listing 3).

Listing 3: Newly added implementation of `DataSource`

```
@Alternative
class PostgresDB implements DataSource {
   private PostgresDBConfig config;

   @Override
   public Connection getConnection() {
      return PostgresDBFactory.create(config);
   }
}
```

Furthermore, the XML file containing the metadata about which implementation to use for injection of `DataSource` is adjusted to use `PostgresDB` (see Listing 4).

Listing 4: Modification of XML metadata for CDI configuration

```
<beans ...>
  <alternatives>
-   <class>InMemoryDB</class>
+   <class>PostgresDB</class>
  </alternatives>
</beans>
```

The result of this change is that any class using an injected object of `DataSource` will now receive an object of type `PostgresDB`, although no changes to the previously existing interfaces and classes were made. Consequently, tests covering any of these affected classes may output different results; RTS techniques can be unsafe, if they fail to select any of those tests. Since none of STARTS, EKSTAZI, or HYRTS analyzes files apart from Java [17], they are potentially unsafe in the presence of such DI-related changes in XML metadata.

2) Safety Violations due to Bean Prioritization: The second example addresses the issue first described by Elsner *et al.* [12], where changes to bean priority can affect test behavior. Here, we use the Spring framework to demonstrate the safety violation; similar scenarios can also be constructed, for example, for CDI. Consider Listing 5, where the original behavior is that an object of the implementation `InMemoryDB` will be created and injected by Spring.

Listing 5: Implementation of `DataSource` before change

```
class InMemoryDB implements DataSource {
   @Override
   public Connection getConnection() {
      return InMemoryDBFactory.create();
   }
}

@Configuration
class InMemoryDBBeanConfig {
   @Bean
   DataSource inMemoryDBBean() {
      return new InMemoryDB();
   }
}
```

Again, suppose another implementation `PostgresDB` is added and prioritized using the `@Primary` source code annotation (see Listing 6). All classes that use a dynamically injected object of `DataSource` will now receive an object of type `PostgresDB`.

Listing 6: Addition of bean with higher priority

```
class PostgresDB implements DataSource {
   @Override
   public Connection getConnection() {
      return PostgresDBFactory.create();
   }
}

@Configuration
class PostgresDBBeanConfig {
   @Bean
   @Primary
   DataSource postgresDBBean() {
      return new PostgresDB();
   }
}
```

This demonstrates that it is possible to introduce and inject beans with higher priority while not changing any existing classes or methods. As pointed out by Elsner *et al.* [12], no code that was present in the old revision may have changed and, thus, the class containing the prioritized bean cannot be present in test execution traces from previous runs. Therefore, existing dynamic RTS techniques such as EKSTAZI or HYRTS are inherently incapable of recognizing such changes, since they can only use test execution traces from the previous run for selecting tests. Although it would be conceptually possible to recognize such changes through static analysis, the static RTS tool STARTS suffers similar limitations, since the new classes `PostgresDB` or `PostgresDBBeanConfig` are not statically referenced anywhere.

III. DEPENDENCY INJECTION AWARE TEST SELECTION

To improve RTS safety in the presence of DI, we present DIRTS, a DI-aware RTS tool. We have alluded to why dynamic RTS techniques reveal an inherent weakness when confronted with DI-related changes (see Sec. II). Therefore, we develop DIRTS on top of static source code analysis using the popular open-source library JAVAPARSER[6]. For the design of DIRTS, we build upon ideas from contemporary RTS tools and adapt them for DI-related changes.

A. Design Decisions

1) Dependency Graph Granularity: We implement and partially adapt two existing approaches differing in *dependency granularity* of code entities in the graph. For class-level analysis, we choose the approach from AUTORTS by Öqvist *et al.* [14], whereas for method-level analysis we use the approach from METHSRTS by Legunsen *et al.* [7]. As such, DIRTS can perform both class-level or method-level dependency analysis, but—similar to AUTORTS, STARTS, and METHSRTS—performs selection at class-level, *e.g.,* JUnit test suites. Since both original approaches are DI-unaware, we extend them to account for dependencies introduced by DI (see Sec. III-B).

2) Analyzing Source Code Instead of Bytecode: As Öqvist *et al.* [14] and Elsner *et al.* [12] point out, if analysis does not require compiled Java bytecode, valuable build time can be saved, since only those code modules that are changed or required for testing need to be compiled. DIRTS therefore employs lightweight source code analysis using JAVAPARSER. When integrated with Maven (DIRTS is available as a Maven plugin), DIRTS can enumerate all affected Maven modules in addition to the selected tests.

3) Adjusting RTS Phases: State-of-the-art RTS tools operate in different phases [8]–[10], [15]. The *analysis* phase analyzes changes and selects tests which run during the *execution* phase. During the *collection* phase, information on dependencies is gathered for the *analysis* in the next run. By collecting dependencies during the *current* execution before executing the tests, we eliminate the problem of not recognizing dependencies from code that has just been added (see Listing 6).

4) DI Extensions: To demonstrate the concept of DI-aware RTS, we implement DIRTS to support three of the most popular DI frameworks for Java: Spring, Guice, and CDI. We chose these frameworks based on findings from prior studies [12], [18], [21]. In Sec. III-B3, we describe how DIRTS parses cross-language links to XML and analyzes source code annotations for these selected DI frameworks.

[6]JAVAPARSER: https://javaparser.org

B. Dependency Analysis

Objects injected via DI may not always be directly injected into test classes. They may be injected into collaborators or the code under test, possibly with several layers of indirection. Therefore, it is not sufficient to only identify edges based on DI. We require a way to track modified code involved in DI transitively in the dependency graph. As previously described, DIRTS supports two strategies for dependency analysis.

1) Class-level RTS – Extending AUTORTS*:* For our class-level implementation, we extend the DI-unaware extraction-based RTS approach proposed by Öqvist *et al.* [14].

Öqvist *et al.* [14] create edges in the dependency graph based on (1) inheritance relations, *i.e.,* using the keywords `extends` and `implements`, (2) constructor invocations using `new`, and (3) calls to `static` methods and access and assignment of `static` variables. These generated edges are directed from the referencing or inheriting class to the invoked, accessed, or inherited class [14]. Because DI generally replaces constructor invocation, AUTORTS needs to be extended for DI mechanisms as we show in Sec. III-B3. We formalize the class-level dependency graph of DIRTS in Def. III.1 (inspired by Orso *et al.* [27] and Legunsen *et al.* [7]).

Definition III.1. The dependency graph created by class-level DIRTS can be seen as a tuple of nodes and edges

$$\langle N, E_{extends} \cup E_{implements} \cup E_{new} \cup E_{static} \cup E_{DI} \rangle$$

where

- N is the set of nodes, in this case representing *classes*, *interfaces*, and *enumerations*
- $E_{extends} \subseteq N \times N$:
  $\langle n_1, n_2 \rangle \in E_{extends}$ if n_1 extends n_2
- $E_{implements} \subseteq N \times N$:
  $\langle n_1, n_2 \rangle \in E_{implements}$ if n_1 implements interface n_2
- $E_{new} \subseteq N \times N$:
  $\langle n_1, n_2 \rangle \in E_{new}$ if n_1 explicitly invokes a constructor of n_2
- $E_{static} \subseteq N \times N$:
  $\langle n_1, n_2 \rangle \in E_{static}$ if n_1 accesses or assigns a static variable or calls a static method in n_2 or if n_1 references an enum constant declared in n_2
- $E_{DI} \subseteq N \times N$:
  edges created through DI are introduced in Sec. III-B3

2) Method-level RTS – Extending METHSRTS*:* For method-level DIRTS, we use METHSRTS by Legunsen *et al.* [7] as a basis, which constructs the edges based on method call graphs. A formalization of our method-level dependency graph is provided in Def. III.2.

The dependency graph of METHSRTS actually uses class-level nodes, *e.g.,* classes or interfaces, such that it does not specifically need to account for dynamic dispatch [7]. The method-level version of DIRTS, in contrast, considers more fine-grained definitions, *e.g.,* methods or constructors, as nodes in the graph. Consequently, by splitting up classes into multiple method nodes, we require additional edges to account for methods called via dynamic dispatch ($E_{inheritance}$ in Def. III.2).

Legunsen *et al.* [7] further report safety violations using their method-call-based dependency graphs, even when not confronted with DI. We address a potential cause of such safety violations by further adding edges to account for accessed and assigned fields ($E_{fieldAccess}$ in Def. III.2).

Definition III.2. The dependency graph created by method-level DIRTS can be seen as a tuple of nodes and edges

$$\langle N, E_{delegation} \cup E_{inheritance} \cup E_{fieldAccess} \cup E_{DI} \rangle$$

where

- N is the set of nodes, in this case representing *methods*, *constructors*, *fields*, *enum constants*, and *initializers*
- $E_{delegation} \subseteq N \times N$:
  $\langle n_1, n_2 \rangle \in E_{delegation}$ if n_2 represents a method or a constructor and n_1 calls n_2
- $E_{inheritance} \subseteq N \times N$:
  $\langle n_1, n_2 \rangle \in E_{inheritance}$ if n_1 and n_2 represent methods and n_2 may be executed via dynamic dispatch when n_1 is called
- $E_{fieldAccess} \subseteq N \times N$:
  $\langle n_1, n_2 \rangle \in E_{fieldAccess}$ if n_2 represents a field accessed by n_1 or if n_1 represents a field assigned to by n_2
- $E_{DI} \subseteq N \times N$:
  edges created through DI are introduced in Sec. III-B3

3) DI-Aware Extensions: To account for DI-related changes and contexts, DIRTS supports the frameworks Spring, Guice, and CDI. Respective extensions to the dependency graphs are conceptually similar across frameworks. Since some DI frameworks harness cross-language links to XML for bean definition, we need to handle these situations accordingly.

a) Extending the Dependency Graph: Similar to the approach by Shi *et al.* [18] for making static RTS safe for reflection in Java, we extend the dependency graph by adding further edges. The idea is to close gaps in the edge relation for objects instantiated and managed through DI. For this purpose, we keep track of all definitions of beans and all injection points in the entire program. The term *injection point* refers to any code entity that specifies a variable that could be injected via DI. Edges are created from a node that represents an injection point to the node that represents a bean, which may be injected into a variable at this injection point. To do so, we extract the *type* of the bean and more framework-specific information such as its *name* or custom annotations called *qualifier annotations* used for identifying beans. Then, we find all injection points that a bean can be injected into based on this information and create edges.

Listing 7: An injection point for `DataSource` in Spring
```
public class DatabaseConnectionTest {
   @Autowired
   private DataSource dataSource;
   // ...
}
```

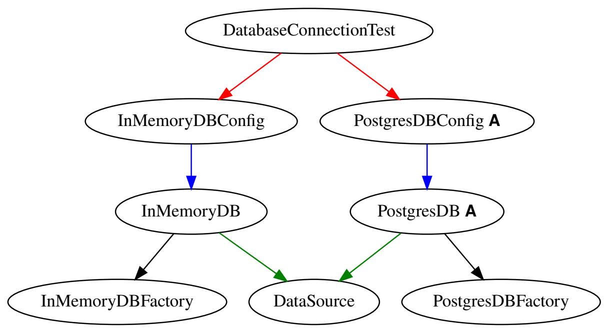

Fig. 1: Dependency graph created by class-level DIRTS, when applied to the example in Listings 5, 6, and 7

In the example from Listings 5 and 6, the two methods `inMemoryDBBean` and `postgresDBBean` would be recognized as beans of *type* `DataSource`, both having the *name* of the respective method (specific to Spring). The field `dataSource`, depicted in Listing 7 would be a corresponding injection point of the same *type*. The graph generated for this example is depicted in Fig. 1. Because the type of the two beans and the injection point match, the two edges colored in red are created by DIRTS's DI extension for Spring. Edges in green result from `extends` or `implements`, edges in blue represent constructor invocation, and edges in black correspond to `static` as formalized in Def. III.1.

By storing beans and injection points to the file system at the end of each run, we can incrementally update these sets while reanalyzing only classes that have changed.

Occasionally, the syntactical elements representing beans do not suit the desired granularity. Consider a bean represented by a whole class that is processed by the method-level algorithm. Creating an edge to the node representing this class is not possible, since the dependency graph used by the method-level approach does not contain nodes of entire classes. Instead, we add edges to the method responsible for instantiating the bean, *i.e.,* the constructors, that are part of the dependency graph.

Similarly, it is not possible to add edges to a method in the graph of the class-level version. In this case, we simply create an edge to the node representing the class that declares this method. The same holds true for fields, as can be seen in Listing 7 and Fig. 1.

b) XML Cross-language Links: Since JAVAPARSER only analyzes Java, we extend DIRTS by an XML parser to find (Spring) beans defined in XML files and add them to the dependency graph. Nodes representing beans in XML are assigned outgoing edges that point to other nodes which represent code entities referenced in the definition of the bean. Ingoing edges are created as described previously.

In CDI, beans can be enabled for injection in the so-called *bean archive*, which can be found in the file `META_INF/beans.xml` inside a Java archive (JAR) [24]. Again, we leverage an XML parser to analyze these cross-language links to XML code. To account for activation and deactivation of beans in XML by adding or removing entries in the `<alternative>`-section of the bean archive [24], we add a custom node for each such entry. In the case of class-level DIRTS, we create an edge starting at the node of the annotated class, pointing towards the node representing the XML entry. For the method-level dependency analysis, we use the constructor node instead. This way, the code representing the bean has a transitive dependency on the XML entry that enables it to be injected.

C. Test Selection

When DIRTS is invoked on a new revision of the code, it first constructs a dependency graph for the new revision. We harness an incremental graph builder algorithm based on file checksums to only analyze the difference to the old revision's dependency graph. The two graphs—representing old and new revision—are combined into a new graph called the *modification graph*. It contains all nodes and edges from both input graphs as well as the modification status of every node. This status can be *added*, *removed*, *modified*, or *unmodified*.

DIRTS supports two test selection algorithms, depending on whether it is used as a *standalone* RTS solution or as a DI safety extension for another RTS tool.

1) Standalone: For selecting tests in standalone mode, we use an approach similar to the one introduced by Öqvist *et al.* [14]: using the reversed edges to determine the tests that reach modified nodes.

DIRTS uses Algorithm 1 to find all nodes that transitively reach any of the modified nodes S, *i.e.,* the set of all nodes with a modification status of *added*, *removed*, or *modified*. These nodes are then filtered to extract those nodes representing tests (not part of Algorithm 1).

Algorithm 1 Calculates all nodes reachable in the transitive closure of the reversed edges — adapted from [14]

```
Require: $N$ nodes in modification graph
Require: $E \subseteq N \times N$ edges
Require: $S \subseteq N$ modified nodes
  procedure REVERSEREACHABLENODES($S, E$)
    $nodesToVisit \leftarrow \{s \in S\}$
    $reachableNodes \leftarrow \{\}$
    while $\neg\, nodesToVisit.empty()$ do
      $n \leftarrow nodesToVisit.next()$
      $reachableNodes \overset{+}{\leftarrow} n$
      for all $o$ with $\langle o, n \rangle \in E$ do
        if $\neg\, o \in reachableNodes$ then
          $nodesToVisit \overset{+}{\leftarrow} o$
        end if
      end for
    end while
    return $reachableNodes$
  end procedure
```

2) DI-aware Extension for Other RTS Tools: To restrict the selected tests to just those affected by DI, we use a slightly different approach shown in Algorithm 3. The idea is to consider only those nodes that reach a modified node with at least one edge resulting from DI in any path between both nodes. We accomplish this by first collecting all nodes that are at the end of such an edge, *i.e.,* nodes representing injected beans. Then, we determine whether each of them reaches a modified node in the transitive closure of the graph via depth-first search (see Algorithm 2). All nodes that fulfill this condition are added to a set R which is then used to invoke Algorithm 1 again. The resulting set of nodes depicts all nodes transitively affected through DI-related edges. Since we are only interested in the affected tests, we filter the nodes to those nodes representing a test.

Algorithm 2 Depth-first search to decide whether a node transitively reaches any modified nodes

```
Require: $E \subseteq N \times N$ edges in modification graph
Require: $N^* \subseteq N$ modified nodes
Require: $n \in N$
  procedure REACHESMODIFIEDNODE($n, E, N^*, visited$)
    if $n \in N^*$ then
      return $true$
    else
      $visited \xleftarrow{+} n$
      for all $\langle n, o\rangle \in E \wedge \neg o \in visited$ do
        if REACHESMODIFIEDNODE($o, E, N^*, visited$) then
          return $true$
        end if
      end for
      return $false$
    end if
  end procedure
```

Algorithm 3 Calculates tests affected by DI-related changes

```
Require: $\langle N, N^*, E\rangle$ modification graph with nodes $N$, modified nodes $N^*$, and edges $E \subseteq N \times N$
Require: $E^* \subseteq E$ edges resulting from DI
Require: $T \subseteq N$ tests
  procedure AFFECTEDBYEDGETYPE($\langle N, N^*, E\rangle, E^*, T$)
    $R \leftarrow \{\}$
    for all $\langle n_1, n_2\rangle \in E^*$ do
      if REACHMODIFIEDNODES($n_2, E, N^*, \{\}$) then
        $R \xleftarrow{+} n_1$
      end if
    end for
    return REVERSEREACHABLENODES($R, E$) $\cap$ $T$
  end procedure
```

D. Integration with Maven Surefire

Similar to EKSTAZI [10] and STARTS [8], we integrate DIRTS with Maven Surefire [28], the testing plugin of Maven, one of the most widespread build management tools for Java projects. In the following, we give examples on how to use DIRTS with Maven to (1) select tests in standalone mode, (2) in combination with other RTS tools, and (3) for selective compilation.

1) DIRTS Standalone: If DIRTS is used in standalone mode (see Sec. III-C1), it outputs a text file containing the list of all excluded test suites. In case Maven Surefire's `excludesFile` property is set, DIRTS will write to this file while acknowledging manually excluded tests inside the file. Listing 8 shows how DIRTS can be invoked in standalone mode before running tests.

Listing 8: Usage in standalone mode

```
$ mvn dirts:class_level_select test
    -Dstandalone
```

2) DIRTS Extension: Using the mechanism to select only the tests affected by DI (see Sec. III-C2), DIRTS can serve as an extension to other RTS tools that aims to correct safety violations related to DI. Our objective is to exclude a test only if both RTS tools agree that it is not affected by the change and does not need to be executed. This is the default behavior of DIRTS and can be used as depicted in Listing 9.

Listing 9: Usage in non-standalone mode

```
$ ... # invoke other RTS tool
$ mvn dirts:class_level_select test
```

RTS tools such as EKSTAZI or STARTS can be configured to output their excluded tests to the `excludesFile` property of Maven Surefire. Thus, DIRTS assumes that there are already some entries in the `excludesFile` and every test which is not excluded has been selected to run by the other RTS tool. DIRTS walks through the list of already excluded tests and comments out all lines representing tests that have been selected by DIRTS. This way, we ensure that the union of the selected tests is executed, by retaining only the intersection of the excluded tests.

3) Saving Build Time: To save build time, as proposed by Öqvist *et al.* [14] and Elsner *et al.* [12], DIRTS is able to generate a list of all Maven modules with at least one affected test, if executed on the outermost module. Compilation and testing can then be restricted to only those modules, as shown in Listing 10.

Listing 10: Usage to save build time

```
$ mvn dirts:class_level_select -Dstandalone
$ mvn -am -pl "$(cat .dirts/
    affected_modules)" test
```

IV. EVALUATION

To evaluate DIRTS, we perform an empirical study on open-source projects that make use of DI. Thereby, we seek to answer the following research questions (RQs):

- **RQ$_1$**: How often do DI-related changes affect tests and yield RTS safety violations?
- **RQ$_2$**: How does DIRTS compare to STARTS and retest-all in terms of test selection ratio and end-to-end execution time?

A. Experimental Setup

1) Study Objects: To evaluate DIRTS, we searched for popular open-source projects written in Java with a substantial commit history on GitHub.

Since there is no point in evaluating DIRTS on projects that do not make use of DI, we manually checked that enough tokens that indicate the use of DI could be found in the code on the latest commit (*e.g.,* `@Bean` for Spring). Because both DIRTS and STARTS come as Maven plugins, we only considered projects that use Maven. Furthermore, these projects need to run on Java Development Kit (JDK) 8 (required by STARTS) and JDK 11 (required by DIRTS), which limited the set of suitable projects. Because EKSTAZI and HYRTS revealed compatibility issues with projects using JUnit 5 [12], we did not consider these dynamic RTS tools in the evaluation. We further did not consider AUTORTS, since it is not available as a Maven plugin. Overall, we included projects that (1) use Maven, (2) rely on DI, using one of the three supported frameworks, (3) support JDK 8 and 11, (4) use JUnit 4 or 5 as testing framework, and (5) do not have any other conflicting characteristics (*e.g.,* code written in Kotlin that would not be analyzed by DIRTS). The projects we use in the evaluation are listed in Table I.

TABLE I: Open-source projects included in evaluation

ID	Name	LOC	# Java Files	# Test Suites	# Commits
			Spring		
S1	rocketmq-dashboard	8,000	120	14	15
S2	infovore	9,000	150	35	30
S3	spring-cloud-aws	23,000	340	95	30
S4	J-MR-Tp	9,000	190	18	22
			Guice		
G1	apollo	11,000	172	20	16
G2	bobcat	17,000	440	40	30
G3	barge	12,000	80	10	25
			CDI		
C1	weld-testing	4,000	110	50	30
C2	smallrye-reactive-messaging	35,000	570	110	30

2) Study Design: As formulated in RQ$_1$, we are particularly interested in DI-related changes. Therefore, we walk through the commit history of the main development branch (*e.g., master*) for each project in random order to find relevant commits for our evaluation. To check whether a commit introduced DI-related changes, we search for tokens relevant for any of the DI frameworks in the changed lines of the commit. The exact patterns we use are part of our supplemental material[7]. This filtering procedure ensures that at the time of the commit, the project does in fact use DI mechanisms. Before a commit was included in the evaluation, we checked if it can be compiled with both JDK 8 and JDK 11 and ran all tests for

[7]Supplemental material: https://github.com/tum-i4/dirts/tree/evaluation

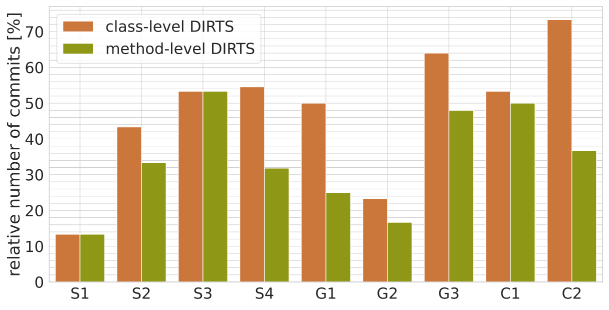

Fig. 2: Ratio of commits with at least one test affected by change related to DI

the project using Maven Surefire. The initial goal was to find 30 such commits per project. However, on certain projects, we were not able to find enough suitable commits, *i.e.,* commits with relevant tokens that could be compiled and tested. On average, we checked 305 commits per project (2,741 in total). The final number of commits for each project, alongside the average lines of code (LOC), number of Java files, and test suites can be found in Table I. For reproducibility, we include a list containing hashes of all analyzed commits for every project, in our supplemental material.

3) Study Procedure: For each project, we step through each of the included commits and execute *retest-all*, STARTS, and DIRTS. In between every run, `mvn clean` is executed, to remove class files of potentially renamed classes and to ensure similar consideration of compilation time when measuring the end-to-end execution time.

a) Retest-all: Retest-all is executed by invoking `mvn test-compile surefire:test`, which first compiles the code and then executes the Maven Surefire test goal.

b) STARTS/DIRTS: To trigger the test selection before running the tests, both STARTS and DIRTS are invoked using the command `mvn test-compile [select-tests] surefire:test`, where `select-tests` is replaced by `starts:run` and `dirts:[class|method]_level_select`, respectively. Alongside, suitable options (*e.g.,* `-DuseSpringExtension`) are provided to activate an DI extension for the project. DIRTS is executed in standalone and non-standalone mode with method- and class-level RTS.

Among other data such as the console output, we collect the end-to-end execution time for each commit and which tests are executed by Maven Surefire.

B. Results

1) RQ$_1$ – Affected Tests By DI-related Changes and Safety Violations: To report the number of DI-related changes and safety violations, we only consider results of the non-standalone version of DIRTS. Fig. 2 shows the ratio of commits on which DIRTS identified at least one test as affected based on edges resulting from DI.

On average across all 9 projects, class-level DIRTS found affected tests on 49.1% of the considered commits. For method-level DIRTS, this ratio is 36%. The two approaches over-

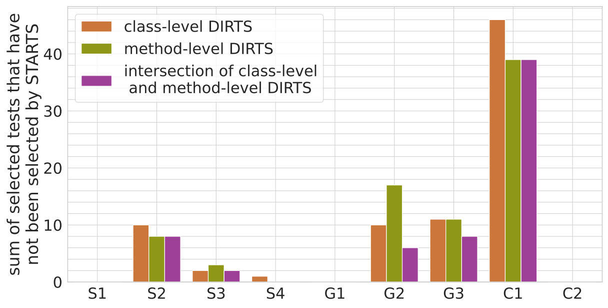

Fig. 3: Sum of all tests DIRTS selected based on DI that STARTS did not select

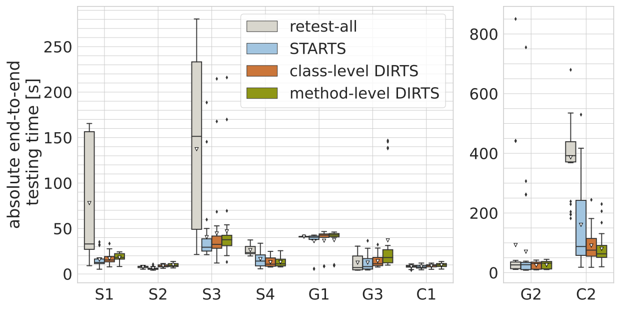

Fig. 4: Distribution of absolute end-to-end execution time across analyzed commits

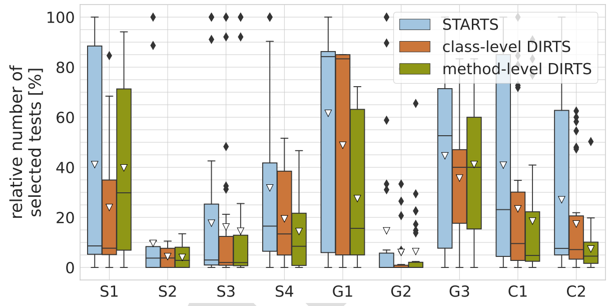

Fig. 5: Distribution of relative number of selected tests compared to retest-all across analyzed commits

approximate dependencies in different ways, which is why these numbers may differ. To filter out over-approximation on either level, we can restrict the analysis to the intersection of tests selected by both approaches, based on DI: for 33.3% of commits, tests have been selected by both approaches due to DI. Even though the number of tests affected through DI varies depending on the project, we can conclude that test-related usage of DI is in fact present in open-source projects as changes lead to tests being affected. Since our evaluation only considered commits that included tokens related to DI (see Sec. IV-A2), this number may well be lower regarding all commits inside a project. Notably, state-of-the-art RTS tools will not necessarily be unsafe in all commits from Fig. 2; it is the upper bound of commits where they might be unsafe.

To get an impression on how likely it is that the tests identified as affected through DI-related edges are missed by state-of-the-art RTS-tools, we also recorded the tests selected by STARTS. This refers to the second part of RQ$_1$. Fig. 3 depicts the number of tests which DIRTS identified as affected by DI-related changes that STARTS did not select, added up over all analyzed commits.

The number of affected tests missed by STARTS is low on average in most projects, but we observe some outliers. Since DIRTS slightly over-approximates dependencies induced by DI (see Sec. III-B3), DIRTS may select tests where run-time behavior is not actually affected by the changes. Therefore, we manually inspect each commit where DIRTS selected tests that STARTS did not select to identify actual cases of unsafe behavior. We thereby confirm 7 commits (3.1%) where STARTS failed to select tests that are certainly affected by changes related to DI. This means that in certain situations—two of which we break down in detail in Sec. IV-C2—DIRTS detected and corrected safety violations related to DI in STARTS.

RQ$_1$ *We find that in 33.3% of analyzed commits DI-related changes affect tests and* DIRTS *identifies unsafe behavior of* STARTS *in 3.1% (7) of commits.*

2) RQ$_2$ – Cost-Effectiveness of DIRTS *vs.* STARTS*:* The goal of RTS is to reduce regression testing effort [7], [9]. Fig. 4 shows the distribution of the absolute end-to-end execution time of retest-all, STARTS, and DIRTS for all analyzed commits.

In 6 of 9 projects, DIRTS reduced the average end-to-end execution time compared to retest-all (including the three largest projects *C2*, *S3*, and *G2*). Thus, the analysis overhead of DIRTS pays off, if there are enough tests, which can be excluded.

STARTS reduced the average end-to-end time in 8 of 9 projects. The fact that STARTS is faster than DIRTS is not surprising, since it uses the constant pool in the class files [8], while DIRTS has to analyze source code to obtain fully qualified names. Still, the average end-to-end time of class-level DIRTS was lower than for STARTS in 3 projects.

As shown in Fig. 5, the overall number of tests selected by DIRTS is lower compared to STARTS. Considering the class-level approach, this is because the algorithm of AUTORTS, which we chose as a basis, does not over-approximate as much as STARTS. As Öqvist *et al.* [14] point out, if a class is just referenced, that does not necessarily mean that code from this class is loaded and traversed during execution of a test. With STARTS, however, in addition to inheritance, any direct use of a class leads to a dependency edge being created [8]. Class-level DIRTS is therefore more precise than STARTS, but following Öqvist *et al.* [14] without compromising safety. For method-level DIRTS, the number of selected tests may be lower due to the finer granularity. Although Legunsen *et al.* [7] found unsafe behavior in METHSRTS, we expect DIRTS to be safer, as we corrected potential safety violations (see Sec. III-B2).

RQ$_2$ *We find that on average across all commits class-level* DIRTS *selected 19.6% of tests and reduced end-to-end execution time by 17.6% compared to retest-all. Method-level* DIRTS *selected only 16.8% of tests, but increased execution time by 35.8% due to higher analysis overhead.*

C. Discussion

For RQ$_1$, we reported cases where DIRTS selected tests that STARTS did not select. In the following, we discuss the root causes for these cases which we identified by manually inspecting the corresponding commits and selected tests.

1) Over-Approximation: A known problem of static RTS is the risk of imprecision, because dependencies are overestimated [7], [17]. According to their inventors, cases of imprecision can be found in both STARTS [8] and AUTORTS [14]. We find similar issues concerning the DI-aware extensions, because dependencies induced through DI mechanisms can be overestimated by DIRTS. Therefore, some tests that DIRTS marked as affected by DI-related changes actually may not encounter a change in runtime behavior.

2) Unsafe Behavior in STARTS*:* Nonetheless, we find 8 tests in 7 commits that STARTS did not select, where it is clear that a DI-related change may influence the test results and the respective tests need to be executed again. 5 of these commits were found in the project `infovore` (*S2*), 1 in `spring-cloud-aws` (*S3*) and 1 in `weld-testing` (*C1*). The former two projects use Spring, whereas the latter one relies on CDI. This means that for the project `infovore`, 16.7% of the analyzed commits revealed safety issues in STARTS. Our findings confirm that using STARTS in the context of DI via Spring or CDI imposes the risk of missing a regression. In the following, we present code excerpts for two of the identified commits, explain why tests are affected by the DI-related changes, and reason why STARTS did not select them.

a) Configuration in XML in Spring: The first commit is from the project `infovore`[8]: it adds an entry to a list, serving as constructor argument of an XML-defined bean, which is in turn used as constructor parameter of a Spring bean (see Listing 11). The bean (`com.ontology2.haruhi.flows.SpringFlow`) is injected into the test suite, as illustrated in Listing 12.

Listing 11: Modified bean in `applicationContext.xml`

```
<bean name="basekbNowFlow" class="com.
    ontology2.haruhi.flows.SpringFlow">
<constructor-arg>
  <list>
    <bean class="com.ontology2.haruhi.flows.
        JobStep">
      <constructor-arg>
        <list>
+         <value>'run'</value>
          <value>'freebaseRDFPrefilter'</value>
          <value>pos[0]+'freebase-rdf-'+pos
              [1]+'/'</value>
          <value>tmpDir+'preprocessed/'+pos
              [1]+'/'</value>
        </list>
      </constructor-arg>
    </bean>
  </list>
</constructor-arg>
</bean>
```

Listing 12: Test using the XML Spring bean

```
@ContextConfiguration({
  "../shell/applicationContext.xml",
  "../shell/testDefaults.xml"})
// ...
public class TestFlowBeans {
    @Autowired SpringFlow basekbNowFlow;
    // ...
}
```

Since the injected bean is used later in a test method, this test clearly is affected by the changes. STARTS failed to select this test because it does not analyze XML files [8], [17].

b) AutoConfiguration in CDI: The single test from `weld-testing`[9], where we found unsafe behavior in STARTS uses custom annotations to configure the injection. The code corresponding to this test can be found in Listing 13. Even though the class of the changed bean occurs once in the code of the affected test, STARTS does not succeed in recognizing this as a dependency. It seems that STARTS does not consider classes referenced in the parameters of annotations. The annotation that has been removed from the class `V8` refers to the scoping mechanism of CDI and thus, this removal may in fact alter the run-time behavior of tests.

Listing 13: Modified class and test that depends on it

```
@EnableAutoWeld
@AddBeanClasses(V8.class)
public class AddBeanClassesTest {
    @Inject private Engine engine;
    // ...
}

- @ApplicationScoped
public class V8 implements Engine,
     Serializable {
    // ...
}
```

D. Threats to Validity

1) Internal Threats: Internal threats to validity include the risk of defects in our implementation or misconfiguration in the evaluation infrastructure. We therefore wrote unit tests and checked our evaluation results for consistency with prior research. Additionally, we implemented sample projects for all three supported DI frameworks to ensure that at least all important use cases are covered.

[8]`infovore` (S2): c2d27d8, test: TestFlowBeans

[9]`weld-testing` (C1): 7243c80, test: AddBeanClassesTest

2) External Threats: External threats include the fact that we only analyzed commits that introduced changes related to DI, to ensure that DI is used at these commits and increase the odds to find tests that are affected by changes related to DI. Hence, the evaluation results might not be representative across all commits of these projects. Further external threats may be imposed by the restricted selection of our study objects. Those projects had to be evaluated on both JDK 8 (STARTS) and JDK 11 (DIRTS), which limited the possible choices. We tried to mitigate this threat by choosing projects from different domains. Considering the generality towards other DI frameworks, we emphasize that our findings depend on the frameworks we considered. Other frameworks may use entirely different mechanisms on injection that may be easier to cope with by state-of-the-art RTS tools such as STARTS.

V. RELATED WORK

In this paper, we investigate how DI can affect the safety of Java RTS tools. Below, we list related work that presents RTS techniques for Java software and their safety assessment.

Gligoric *et al.* [9], [10] develop EKSTAZI, the first dynamic file-level RTS tool for Java software. EKSTAZI instruments Java bytecode files at run-time and thereby collects per-test execution traces. To compute the set of affected tests, EKSTAZI uses smart file checksums and reduced end-to-end testing time on average by 32% when evaluated on 32 open-source projects.

While EKSTAZI traces class file accesses, HYRTS presented by Zhang [15], collects more fine-grained information about dependencies at the level of methods and derives class level dependencies from this information. Using this hybrid information, HYRTS aims to increase precision when only parts of a file are changed, while having low analysis overhead when the whole file has changed. Similar to EKSTAZI, HYRTS uses smart checksums to detect changes and outperformed EKSTAZI in terms of selection ratio across 32 open-source projects.

Legunsen *et al.* [7] propose two static approaches for RTS in Java software. The first approach, CLASSSRTS, calculates dependencies of class files based on the so-called Intertype Relation Graph (IRG). In contrast, the second approach called METHSRTS identifies links between methods using the method call graph. By computing the transitive closure of each test, they determine the tests affected by changes. Although METHSRTS creates edges based on method calls, both approaches detect changes at the level of classes. While CLASSSRTS achieves similar performance as EKSTAZI, METH-SRTS exhibits more overhead and safety violations [7]. One year later, Legunsen *et al.* [8] published an adaptation of CLASSSRTS as the tool STARTS, which does not distinguish between inheritance and reference associations (*i.e.,* direct use).

Öqvist *et al.* [14] propose AUTORTS, a static RTS technique that performs more fine-grained analysis and effectively generates a subset of the edges of the dependency graph maintained by STARTS. Moreover, AUTORTS directly operates on Java source code and therefore, contrary to STARTS, does not require a fully compiled workspace. When evaluated on 5 open-source projects, the average run time with AUTORTS was 13%–63% of retest-all on 4 out of 5 projects. As outlined in Sec. III, the class-level analysis of DIRTS is inspired by AUTORTS.

Static RTS tools such as STARTS can be unsafe if Java projects make use of reflection [8]. Shi *et al.* [18] attempt to solve this issue by evaluating five different static and dynamic analysis techniques to account for reflection. Similar to DIRTS, they add missing edges to the static dependency graph, more precisely the IRG used by STARTS. While all studied techniques show increased safety related to reflection, they jeopardize precision and cost-effectiveness regarding the time needed for testing. In their evaluation, Shi *et al.* further find missing edges in dependency graphs related to the use of DI frameworks.

Zhu *et al.* [17] develop RTSCHECK, a system for automatically assessing RTS tools regarding safety violations, precision, and unexpected behavior. Through RTSCHECK, several sources of safety violations, including changes to external files not related to Java, such as XML files, have been identified.

To account for external file accesses across JVM boundaries, Celik *et al.* [26] propose RTSLINUX, a dynamic file-level RTS tool leveraging system call instrumentation on Linux. Similar to EKSTAZI and STARTS, RTSLINUX uses smart file checksums for selecting tests and therefore requires a readily available compiled workspace. Celik *et al.* evaluate RTSLINUX on 21 Java projects and report savings of on average 53% in test execution time compared to retest-all.

In an industrial study, Elsner *et al.* [12] develop a dynamic RTS technique which combines language-agnostic system call tracing with lightweight static and dynamic analysis. Contrary to previous RTS techniques, their approach is build system aware, that is, it does not need a fully compiled code base to operate and accounts for changes in build system configuration. When evaluating their RTS technique in industry-scale continuous integration (CI) infrastructure, they save on average 50%–63% of pipeline execution time. They further describe potential safety violations of their dynamic RTS technique, *e.g.,* when new dynamically injected beans are added without changing existing code.

To conclude, Shi *et al.* [18] and Elsner *et al.* [12] first identified potential safety violations in RTS when confronted with DI-related changes. In addition, Zhu *et al.* [17] discovered safety violations with changes to external non-Java files, such as XML files, which are commonly used by DI frameworks. Yet, to the best of our knowledge, no prior RTS research exists that investigates or resolves safety violations related to DI.

VI. CONCLUSION

In this paper, we present DIRTS, a novel static RTS tool for Java, which is aware of changes related to DI frameworks. By adding further edges to the dependency graph, DIRTS accounts for dependencies induced by DI mechanics. While DIRTS may be used as a standalone RTS solution, it can also serve as a safety extension to other RTS tools. In an empirical study on 228 commits from 9 open-source projects, we show that DI-related changes affected tests in 33.3% of the analyzed commits. While correcting for safety violations in a contemporary RTS tool in 3.1% of commits, DIRTS proved comparative in efficiency. Since DI frameworks are widely used in Java projects—specifically in the area of web application development—we expect DIRTS to be a relevant extension to existing DI-unaware RTS solutions.

ACKNOWLEDGMENT

This work was partially funded by the German Federal Ministry of Education and Research (BMBF), grant Q-Soft 01IS22001B. The responsibility for this article lies with the authors.

REFERENCES

[1] S. Hundsdorfer, D. Elsner, and A. Pretschner, "Tool Demo Video: DIRTS - Dependency Injection Aware Regression Test Selection," 1 2023. [Online]. Available: https://doi.org/10.6084/m9.figshare.21355095.v1

[2] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: A survey," *Software Testing Verification and Reliability*, vol. 22, pp. 67–120, 2012.

[3] K. F. Fischer, "A test case selection method for the validation of software maintenance modifications," in *Proceedings of International Computer Software and Applications Conference*, 1977, pp. 421–426.

[4] K. Fischer, F. Raji, and A. Chruscicki, "A methodology for retesting modified software," in *Proceedings of the National Telecommunications Conference*, 1981, pp. 1–6.

[5] G. Rothermel and M. J. Harrold, "A safe, efficient regression test selection technique," *ACM Transactions on Software Engineering and Methodology*, vol. 6, p. 173–210, 1997.

[6] ——, "A framework for evaluating regression test selection techniques," in *Proceedings of the International Conference on Software Engineering*, 1994, pp. 201–210.

[7] O. Legunsen, F. Hariri, A. Shi, Y. Lu, L. Zhang, and D. Marinov, "An extensive study of static regression test selection in modern software evolution," in *Proceedings of the International Symposium on Foundations of Software Engineering*, 2016, pp. 583–594.

[8] O. Legunsen, A. Shi, and D. Marinov, "Starts: Static regression test selection," in *Proceedings of the International Conference on Automated Software Engineering*, 2017, pp. 949–954.

[9] M. Gligoric, L. Eloussi, and D. Marinov, "Practical regression test selection with dynamic file dependencies," in *Proceedings of the International Symposium on Software Testing and Analysis*, 2015, pp. 211–222.

[10] ——, "Ekstazi: Lightweight test selection," in *Proceedings of the International Conference on Software Engineering*, 2015, pp. 713–716.

[11] A. Shi, P. Zhao, and D. Marinov, "Understanding and improving regression test selection in continuous integration," in *Proceedings of the International Symposium on Software Reliability Engineering*, 2019, pp. 228–238.

[12] D. Elsner, R. Wuersching, M. Schnappinger, A. Pretschner, M. Graber, R. Dammer, and S. Reimer, "Build system aware multi-language regression test selection in continuous integration," in *Proceedings of the International Conference on Software Engineering: Software Engineering in Practice*, 2022, pp. 87–96.

[13] D. Elsner, D. Bertagnolli, A. Pretschner, and R. Klaus, "Challenges in regression test selection for end-to-end testing of microservice-based software systems," in *Proceedings of the International Conference on Automation of Software Test*, 2022, pp. 1–5.

[14] J. Öqvist, G. Hedin, and B. Magnusson, "Extraction-based regression test selection," in *Proceedings of the International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, 2016, pp. 1–10.

[15] L. Zhang, "Hybrid regression test selection," in *Proceedings of the International Conference on Software Engineering*, 2018, pp. 199–209.

[16] L. Zhang, M. Kim, and S. Khurshid, "Faulttracer: A spectrum-based approach to localizing failure-inducing program edits," *Journal of Software: Evolution and Process*, vol. 25, pp. 1357–1383, 2013.

[17] C. Zhu, O. Legunsen, A. Shi, and M. Gligoric, "A framework for checking regression test selection tools," in *Proceedings of the International Conference on Software Engineering*, 2019, pp. 430–441.

[18] A. Shi, M. Hadzi-Tanovic, L. Zhang, D. Marinov, and O. Legunsen, "Reflection-aware static regression test selection," *Proceedings of the International Conference on Programming Languages*, vol. 3, pp. 1–29, 2019.

[19] L. B. Cuong, V. S. Nguyen, D. A. Nguyen, P. N. Hung, and D. H. Vo, "Jcia: A tool for change impact analysis of java ee applications," in *Advances in Intelligent Systems and Computing*, vol. 672, 2018.

[20] R. Laigner, M. Kalinowski, L. Carvalho, D. Mendonça, and A. Garcia, "Towards a catalog of java dependency injection anti-patterns," in *Proceedings of the Brazilian Symposium on Software Engineering*, 2019, pp. 104–113.

[21] P. Mayer, "A taxonomy of cross-language linking mechanisms in open source frameworks," *Computing*, vol. 99, pp. 701–724, 2017.

[22] S. M. Biju, "Dependency injection for loose coupling of objects," *International Journal of Scientific & Engineering Research*, vol. 3, 5 2012.

[23] L. Saeed, *Introducing Jakarta EE CDI: Contexts and Dependency Injection for Enterprise Java Development*. Apress, 2020.

[24] A. Sabot-Durand, *Contexts and Dependency Injection*, 3rd ed. Jakarta EE, 7 2020.

[25] D. R. Prasanna, *Dependency Injection: Design patterns using Spring and Guice*. Manning Publications, 2009. [Online]. Available: https://learning.oreilly.com/library/view/dependency-injection-design/9781933988559/

[26] A. Celik, M. Vasic, A. Milicevic, and M. Gligoric, "Regression test selection across jvm boundaries," in *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2017, pp. 809–820.

[27] A. Orso, N. Shi, and M. J. Harrold, "Scaling regression testing to large software systems," in *Proceedings of the International Symposium on Foundations of Software Engineering*, 2004, p. 241–251.

[28] A. Maven, "Maven surefire plugin – surefire:test," 2021. [Online]. Available: https://maven.apache.org/surefire/maven-surefire-plugin/test-mojo.html