A Framework for Writing Trigger-Action Todo Comments in Executable Format

Pengyu Nie, Rishabh Rai, Junyi Jessy Li, Sarfraz Khurshid, Raymond J. Mooney, and Milos Gligoric
The University of Texas at Austin (USA)
{pynie@,rishabh.rai@,jessy@austin.,khurshid@ece.,mooney@cs.,gligoric@}utexas.edu

ABSTRACT

Natural language elements, e.g., todo comments, are frequently used to communicate among developers and to describe tasks that need to be performed (actions) when specific conditions hold on artifacts related to the code repository (triggers), e.g., from the Apache Struts project: "remove expectedJDK15 and if() after switching to Java 1.6". As projects evolve, development processes change, and development teams reorganize, these comments, because of their informal nature, frequently become irrelevant or forgotten.

We present the first framework, dubbed TrigIt, to specify trigger-action todo comments in executable format. Thus, actions are executed automatically when triggers evaluate to true. TrigIt specifications are written in the host language (e.g., Java) and are evaluated as part of the build process. The triggers are specified as query statements over abstract syntax trees, abstract representation of build configuration scripts, issue tracking systems, and system clock time. The actions are either notifications to developers or code transformation steps. We implemented TrigIt for the Java programming language and migrated 44 existing trigger-action comments from several popular open-source projects. Evaluation of TrigIt, via a user study, showed that users find TrigIt easy to learn and use. TrigIt has the potential to enforce more discipline in writing and maintaining comments in large code repositories.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**; *Domain specific languages*; *Software evolution.*

KEYWORDS

Todo comments, trigger-action, domain specific languages

ACM Reference Format:
Pengyu Nie, Rishabh Rai, Junyi Jessy Li, Sarfraz Khurshid, Raymond J. Mooney, and Milos Gligoric. 2019. A Framework for Writing Trigger-Action Todo Comments in Executable Format. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19), August 26–30, 2019, Tallinn, Estonia.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3338906.3338965

1 INTRODUCTION

Natural language elements, such as *todo comments*, are frequently used to communicate among developers [35, 46, 49]. Some of these comments document that a developer should perform an *action* if a *trigger* evaluates to true, e.g., from the Apache Wave project [26]:

$$\text{“}\underset{\text{action}}{\underline{\textit{Remove this [line]}}}\ \textit{when}\ \underset{\text{trigger}}{\underline{\textit{HtmlViewImpl implements getAttributes}}}\text{”}$$

We consider those comments where the trigger is a query over artifacts related to the code repositories and actions are either notifications to developers or code modifications. We call these comments *trigger-action comments*.

Although trigger-action comments are ubiquitous [31, 35, 45], they are, like other types of comments, written in natural language. Thus, as projects evolve, development processes change, and development teams reorganize, these comments frequently become irrelevant or forgotten. As an example, consider the following comment from the Apache Gobblin project [18]:

$$\text{“}\underset{\text{action}}{\underline{\textit{Remove [this class]}}}\ \textit{once}\ \underset{\text{trigger}}{\underline{\textit{we commit any other classes}}}\text{”}$$

This comment, followed by an empty class, was included in December 2015 in the `package-info.java` file to force the Javadoc tool to generate documentation for an empty package. Three months later (February 2016) classes were added to the package, but the comment and the empty class in `package-info.java` file were not removed. More than three years later (2019), the comment and the empty class are still in the repository.

Having pending actions, i.e., those actions that should have been done because the triggers evaluate to true, and outdated comments may negatively impact program comprehension and maintenance [35, 45, 46, 49–51]. Additionally, having comments written in an informal way presents a challenge for some software engineering tools, such as refactorings [17, 37, 38], as those tools may not know how to manipulate code snippets and identifiers embedded in comments [44].

Developers in industry have recognized the problem with outdated todo comments and recently developed tools to help with maintenance of todo comments. `imdone` [31] extracts and maintains the list of pending todo comments at one place. `todo_or_die` [45] enables developers to write executable statements that will break the program execution if a todo comment is not addressed by a specific date. The main developer of `todo_or_die` says: "[in the past] the comment did nothing to remind myself or anyone else to actually delete the code [...] this eventually resulted in an actual support incident (long story)" [45].

To further motivate our work, we also reached out to ten developers at large software companies, including Google, Dropbox, Groupon, and Palantir, to ask if they write todo comments and trigger-action comments. Based on eight replies, seven developers said that they have todo comments and trigger-action comments. These developers also highlighted the importance of improving the maintenance of todo comments, e.g., "I'm very tired of half-complete migrations and stale todos" and "we have a lot of todos at [company] that get forgotten until something breaks".

With the goal to enforce more discipline in writing and maintaining comments, we present the first framework, dubbed TrigIt, to specify trigger-action comments in *executable format*; the triggers are evaluated automatically at each build run and actions are taken based on users' specifications. TrigIt introduces a domain specific language (DSL) that can be used to write triggers and actions. Specifically, triggers are written as *query statements* over ASTs, build configuration scripts, issue tracking systems, and system clock time; actions are either *notifications to developers* or *transformation steps over ASTs.* To provide a natural access to the AST elements and improve maintenance, the TrigIt DSL is embedded in the host language. However, the semantics of the language enables the executable trigger-action comments to be evaluated as part of the static program analysis phase (prior to program execution).

We implemented the TrigIt technique in a tool for Java; we use the same name for both the technique and the tool. TrigIt analyzes compiled code and modifies either compiled code, source code, or neither, depending on user-defined actions. TrigIt allows users to force execution of actions, for example during testing or during debugging of TrigIt specifications, without modifying sources. If a user chooses to modify source code with an action, she would be provided with a patch once the action is taken. TrigIt should be integrated in the build process after the compilation step. Ideally, we envision TrigIt being used as a bot that sends a code review with changes whenever a trigger evaluates to true.

We evaluated TrigIt via a user study with 20 participants which showed that users find TrigIt easy to learn and use; most users correctly encoded trigger-action comments in the TrigIt DSL with less than ten minutes of training. Additionally, we evaluated TrigIt by manually migrating 44 existing trigger-action comments to the TrigIt DSL; all the comments are from ten popular open-source projects available on GitHub. In our experiments, we also report the complexity of TrigIt statements measured in terms of the number of tokens in the specifications, as well as the overhead introduced by the tool in the build process. TrigIt does *not* introduce any overhead at runtime. Our results show that TrigIt introduces negligible overhead in the build process, and triggers and actions are short to write (on average 18.0 tokens).

The main contributions of this paper include:

⋆ **Idea**: We introduce an idea to encode trigger-action comments, currently written in natural language, as executable statements in the host language. Having executable trigger-action comments enables their maintenance (e.g., refactoring), testing, and automatic execution of the triggers and actions when artifacts related to the code repository change.

⋆ **Tool**: We implemented our idea in a tool, dubbed TrigIt, for Java. We also developed a Maven plugin to simplify the integration of TrigIt with an existing build system.

⋆ **User study and case studies**: We evaluated TrigIt via a user study with 20 participants, including 6 developers working for large software companies. Additionally, we report our experience in manual migration of existing comments.

⋆ **Dataset**: A byproduct of our work is the first dataset of trigger-action comments and their executable counterparts. Additionally, we manually added various labels on the comments that can be useful in future research projects.

Our tool and the dataset are available at cozy.ece.utexas.edu/trigit.

```
// AbstractStreamingHasher.java
protected AbstractStreamingHasher(int chunkSize, int bufferSize) {
  // TODO(kevinb): check more preconditions
  // (as bufferSize >= chunkSize) if this is ever public
  if (trigItIsPublic())
    checkArgument(bufferSize >= chunkSize);
  checkArgument(bufferSize % chunkSize == 0); ... }
@TrigtItMethod
boolean trigItIsPublic() {
 return TrigIt.getMethod("<init>", int.class, int.class).isPublic();}
```

Figure 1: An example from the `google/guava` project and TrigIt encoding to illustrate a local action.

```
// FreemarkerResultMockedTest.java
void testDynamicAttributesSupport() throws Exception { ...
  // TODO : remove expectedJDK15 and if() after switching to Java 1.6
  if (TrigIt.getJavaVersion().ge(TrigIt.JAVA6)) {
    String expectedJDK16 = "<input type=\"text\" ...";
    assertEquals(expectedJDK16, result);
  } else {
    String expectedJDK15 = "<input type=\"text\" ...";
    String expectedJDK16 = "<input type=\"text\" ...";
    if (result.contains("foo=\"bar\" ..."))
      assertEquals(expectedJDK15, result);
    else
      assertEquals(expectedJDK16, result); } ... }
```

Figure 2: An example from the `apache/struts` project and TrigIt encoding to illustrate a build configuration trigger.

```
// Mapper.java
// TODO: make this protected once Mapper and FieldMapper are merged together
public final String simpleName() { return simpleName; }
@TrigItMethod
void checkMerge() {
  if (!TrigIt.hasClass("Mapper")
       || !TrigIt.hasClass("FieldMapper")) {
    TrigIt.getMethod(simpleName()).setProtected(); } }
```

Figure 3: An example from the `elastic/elasticsearch` project and TrigIt encoding to illustrate a global action.

2 ILLUSTRATIVE EXAMPLES

This section presents several existing trigger-action comments from large open-source projects, and the encoding of these comments in the TrigIt DSL and TrigIt's workflow. We chose the comments such that we can illustrate various aspects of the TrigIt approach.

Figure 1 shows a code snippet from the `google/guava` project [19]. We show the encoding of the executable trigger-action comment within boxes. This comment was added in commit `c92e1c7` (2017-05-31) and is still present as of 2019-06-11 (`bf9e8fa`). In this case, a developer wants to add more precondition checks if the method or constructor becomes `public`. The trigger in TrigIt is encoded as a separate method that returns a `boolean` value, and each TrigIt method needs to have the @TrigItMethod annotation. `trigItIsPublic` finds the constructor and checks its modifiers.

Table 1: Examples of Query Expressions in the TrigIt DSL.

Type	Query Expression	Natural Language Description
AST	`TrigIt.getClasses().findAny("C").isPresent()`	Checks if there is a class with name "C"
Build	`TrigIt.getJavaVersion().ge(TrigIt.JAVA8)`	Checks if the Java version is greater or equal than Java 8
Issue	`TrigIt.isClosed("https://github.com/google/closure-compiler/issues/1897")`	Checks if the issue (specified with URL) is closed
Time	`TrigIt.after("2019-04-02")`	Checks if the system time is after 2019-04-02

The invocation of `trigItIsPublic` is a *guard* for an extra precondition check. It is important to observe that in this case, the action (transformation step) is *local*, i.e., we simply execute extra statements within the method. As this comment is not specific enough, i.e., we do not know all the preconditions that developers would like to check, we could include an extra action to print a warning to developers when the trigger evaluates to true. Recall that triggers are evaluated during the build process *prior* to the execution; this means that methods annotated with @TrigItMethod, `if` statements that guard actions, and statements of either then or else branch are removed by TrigIt prior to the program execution. As stated earlier, ideally TrigIt is integrated in a bot that sends code reviews or pull requests once the trigger evaluates to true.

Figure 2 shows a code snippet from the `apache/struts` project [7], which is a web framework for creating Java web applications. The specified action was performed at commit `a5812bf` (2015-10-06), five months after the trigger evaluated to true. Unlike the previous example, this one illustrates a query statement over the build configuration script. Specifically, the trigger evaluates to true if the current Java version is greater than 1.6. Regarding the action, we guard the modified code in the then branch and the original code is in the else branch.

Finally, Figure 3 shows a code snippet from the `elastic/elasticsearch` project [13], which is a distributed search engine. We use this example to illustrate the *global* code transformation action. In this example, developers want to change the access modifier of a method (`simpleName`) from `public` to `protected` if two classes (`Mapper` and `FieldMapper`) are merged. Although there is no unique way to encode a trigger that checks if two classes are merged, the check can be approximated in several ways. Our approach is to check that one of the classes is no longer available. A better option might be to check that one class is removed while the other one is still present. By knowing the relation between the classes (`FieldMapper` extends `Mapper`) and their usage, we believe that the original developers could provide a more precise trigger. The action specifies that the modifier of the method should be changed to `protected`. Unlike prior examples, the action is global, i.e., impacts code elements outside a method body, and it is expressed as a transformation step over the class AST.

3 TRIGIT TECHNIQUE

This section describes the TrigIt DSL, presents the workflow and the integration with existing build processes, and briefly describes the current implementation.

3.1 Language

Specifications in the TrigIt DSL are written in a subset of Java with slightly modified semantics. The TrigIt DSL consists of three syntactic components: *query expression* for triggers, *action statement* for actions, and *TrigIt method*. A query expression is syntactically

```
package org.trigit.project;
public class ClassModel extends ModelBase {
    public String getName() {...}
    public ModifiersModel getModifiers() {...}
    public List<FieldModel> getFields() {...}
    public List<MethodModel> getMethods() {...}
    public boolean isPublic() {...}
    public boolean isProtected() {...}
    public boolean isPrivate() {...}
    public boolean isPackagePrivate() {...} ... }
```

Figure 4: Part of TrigIt's ClassModel API.

equivalent to a Java expression (that evaluates to boolean). An action statement is syntactically equivalent to a Java statement. A TrigIt method is syntactically equivalent to a Java method declaration (with either boolean or void return value and without arguments). We first describe each component, and then describe how they are combined into TrigIt specifications.

Query expression. Each query expression is a logical expression, such that each operand queries the state of the artifacts related to the code repository via the TrigIt Application Programming Interface (API). The triggers supported in our design include queries over (1) *AST* elements, (2) *build* configuration scripts, (3) *issue* tracking systems, and (4) system clock *time*. Our decision to support these trigger types is based on the most commonly seen types of trigger-action comments that developers write in open-source projects. Table 1 lists the types of triggers as well as one example for each type. Each trigger has to start with an invocation of the TrigIt API.

For the query over AST, a user works with a stream of model classes. Once a developer obtains a stream, the developer can use any standard Java stream operation [6] (e.g., `map`, `filter`, `count`, etc.) to create a query. In case the stream support is not available in the Java version used by the project, the developers may opt for equivalent operations available in the TrigIt API. The result of each query has to be a boolean value. Clearly, as arguments to stream operations, the developer can use model classes, fields, methods, etc. to access AST elements and their properties, including class name, modifier, return type, and others. We only show signatures of a few methods from `ClassModel` in Figure 4, which is a model of a class in the project. TrigIt model classes offer a rich API, and many syntactic sugars are available, e.g., get a model of a method by name; the complete TrigIt API is available on the accompanying web page [54].

Additionally, when constructing queries, a developer can use constant values, as well as field accesses and method invocations; however, the semantics for the latter two differ from the one specified by the Java Language Specification, as we discuss below.

For the last three query types, TrigIt provides an API to get the build configuration, an API to check the status of an issue on an issue tracking system, as well as an API to check if the current system clock time is past a specific date.

Action statement. Similarly to a query expression, each action statement has to start with an invocation of the TrigIt API to obtain a stream of model classes. We currently provide only an API for modifying ASTs of classes but not other artifacts related to the code repository. Unlike the instances of model classes that are available in the query expressions, the instances of model classes available in actions can be both queried and *modified*, i.e., extra API methods are available to specify modifications; this is similar to API's available in IDEs to perform AST rewrites [16]. The expressions used as arguments to stream operations may include constants, field accesses, and method invocations. There are no limits on the number of action statements that can be guarded by a single query expression. For example, if we want to modify an access modifier of a field "f" in the current class, we can write (in a short form) the following action statement:

```
TrigIt.getField(f).setPrivate();
```

TrigIt method. Each TrigIt method should be a Java method that (1) has a return type boolean or void, (2) has no arguments, and (3) is annotated with @TrigItMethod. TrigIt methods that return a boolean value need to have only a single statement: `return ⟨query expression⟩;` that implements the trigger. TrigIt methods that do not return any value (i.e., void) can have multiple statements, but the first statement has to be an if statement, such that the conditional expression is a query expression; Other statements, which are always a part of the then block, are action statements. The following are templates for TrigIt methods with a boolean return value and with a void return type, respectively.

```
@TrigItMethod boolean <modifiers> <name>() {
    return <query expression>; }
@TrigItMethod void <modifiers> <name>() {
    if (<query expression>) { <action statement>* }}
```

TrigIt specifications. A complete TrigIt specification consists of both a trigger part and an action part. We differentiate two types of actions: (1) global and (2) local. We define a *global action* as a sequence of statements that specify modifications to the program structure [59]. These actions modify *out-of-method code elements*, including method signatures, class declarations, etc. Figure 3 from Section 2 illustrates a global action that updates a modifier of a method. A TrigIt specification with a global action is written as a TrigIt method with a void return type, where the query expression is the trigger part and the action statements are the action part.

We define a *local action* as a sequence of statements that should or should not be executed depending on a trigger that guards those statements. We say that these actions modify *in-method* code elements (i.e., a sequence of statements to be executed). Figures 1 and 2 from Section 2 illustrate local actions. A local action is written as an if statement, whose condition is a query expression or a TrigIt method with a boolean return value, and whose then branch is the code that should be executed if the trigger evaluates to true, while the else branch is the code that should not be executed in that case. The following is the general format of a TrigIt specification with a local action:

```
if (<query expression> | <TrigIt method name>()) <statement>
else <statement>
```

if (<expr>) ... → ⊥	[<expr> ≠ <query expression>]
<stmt> → ⊥	[<stmt> ∉ {<action statement>, <if statement>}]
this.f | ClassName.f → "f"	[ClassName ≠ TrigIt]
this.m() | ClassName.m() → "m"	[ClassName ≠ TrigIt]
this.m(arg1, ...) | ClassName.m(arg1, ...) → "m", arg1.type, ...	[ClassName ≠ TrigIt]

Figure 5: Rewrite rules to "prepare" TrigIt methods for the evaluation by the framework.

When writing a local action statement, a developer may opt to use an API call available only in a new version of a library, e.g., `java.util.List.of(...)` from Java 9. However, writing such code could result in compilation errors if project uses Java 8. One approach that the developer can take is to write a method invocation using the reflection mechanism [5].

Semantics. Although evaluation of the TrigIt specifications closely follows Java semantics, there are two main differences. First, all method invocations and field accesses (that do not belong to the TrigIt API) are substituted with the names of methods and fields, i.e., those methods and fields are never invoked or accessed. In case of a method invocation, all the arguments are replaced with their corresponding types. (The TrigIt API uses the argument types to differentiate between overloaded methods.) For example, the code snippet shown above that changes a filed access to private would be modified prior to the evaluation to:

```
TrigIt.getField("f").setPrivate();
```

We made this decision to avoid using strings to refer to a field, method, or class name unless that is absolutely necessary. Our decision will help to keep comments up-to-date with code, e.g., during refactoring, to avoid what researchers call fragile comments [44].

Second, prior to evaluating a TrigIt specification, the class that contains the method is rewritten to remove anything other than the query expressions, action statements (in TrigIt methods), and if statements. This is done to enable evaluation of the executable trigger-action comments without worrying about the environment that is required to execute any piece of code from the project itself. For example, even loading a class may require substantial setup and execution cost if a static block is present. To prepare a class for evaluation with TrigIt, we define a set of rewrite rules, shown in Figure 5. Each rule has the following format:

before → after [condition]

where *before* and *after* are AST elements in Java or an empty string denoted by ⊥; *condition* defines when a rule is applicable. The first two rules remove any statement from a class that is irrelevant for TrigIt specifications. The last three rules rewrite each field access to the name of the accessed field, and method invocation to the name of invoked method and types of its arguments (if any). We recursively apply the rewrite rules on the method until no more rewrite rules can be applied, and then the obtained code can be evaluated by Java.

Rationale. We would like to emphasize that our decision to enable the evaluation of trigger-action comments independently of other code was to keep the separation between production code and comments, to enable the evaluation of comments regardless of the requirements needed for running the project's code, and to avoid the

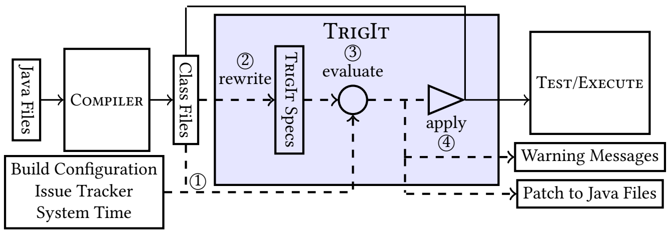


Figure 6: TrigIt's workflow.

$$\begin{aligned}
&\text{<query expression>} \rightarrow \text{evaluate(<query expression>)} && \\
&\text{m()} \rightarrow \text{evaluate(m)} && \text{[m is TrigIt method]} \\
&\text{if (true) <stmt1> else <stmt2>} \rightarrow \text{<stmt1>} && \\
&\text{if (false) <stmt1> else <stmt2>} \rightarrow \text{<stmt2>} && \\
&\text{<modifiers> m() ...} \rightarrow \perp && \text{[@TrigItMethod} \in \text{<modifiers>]}
\end{aligned}$$

Figure 7: Rewrite rules for applying the local actions and removing the TrigIt specifications.

performance overhead of evaluating comments at runtime. As our decisions are inspired by examples found in open-source projects and feedback from several developers working in industry, some of these decisions might need to be revisited in the future to support the encoding of comments as executable statements for a broader class of comments.

3.2 Workflow

Figure 6 shows TrigIt's workflow. TrigIt interposes between the compiler and (test) execution. The first step to use TrigIt is to encode existing trigger-action comments as executable TrigIt specifications. Once a project is compiled, the query expressions, action statements and TrigIt methods, are part of the resulting classfiles. Having executable trigger-action comments checked by the compiler is one advantage over informally written comments.

TrigIt accepts the compiled classfiles and works in four steps. First (step ① in Figure 6), TrigIt processes all the classfiles from the project to build the intermediate AST representation to be used by the query expressions and action statements; TrigIt also retrieves build configurations, issue status (from an issue tracker) and system time lazily upon request by the query expressions. Next (step ②), TrigIt modifies classfiles, based on the rewrite rules in Figure 5, to prepare the TrigIt specifications for evaluation. The modified classfiles are never stored on disk, unless a developer specifies the debug option, but they are only available in-memory and they are dynamically loaded [4]. Then (step ③), TrigIt evaluates each query expression and boolean TrigIt method in a non-deterministic order. We discuss potential dependencies between TrigIt methods in Section 7. If the project being built requires a Java version prior to Java 8, which is currently required for TrigIt execution, TrigIt methods are evaluated by spawning an external process.

Finally (step ④), TrigIt takes actions and applies changes depending on the evaluation results. If a query expression or a boolean TrigIt method evaluates to true, there are three possible outcomes. First, TrigIt can notify a developer with the list of triggers that hold, without executing any action. In addition to printing which triggers hold, TrigIt also includes a short explanation that justifies the outcome of the trigger (e.g., "Java version "1.8" greater than "1.7"; at pom.xml:77").

Second, TrigIt can rewrite the classfiles on disk to apply the actions (both global and local) guarded by those triggers evaluated to true. For global actions, the action statements are executed to modify the ASTs. Then, TrigIt applies a set of rewrite rules in Figure 7 to apply the local actions and to clean the TrigIt specifications from the classfiles; we omit the condition in a rewrite rule if the rule always applies. The first two rules inline the evaluation results of query expressions and boolean TrigIt methods. The next two rules only keep the correct branches for local actions, depending on the evaluation results. The last rule cleans all TrigIt methods from the classfiles. The resulting classfiles for test or execution contain no TrigIt specifications, thus TrigIt introduces no runtime overhead.

Finally, TrigIt can be configured to create a patch for the source code, which developers may inspect, modify, and apply. The configuration provided by the user determines what option is taken. The configuration options are not mutually exclusive. As mentioned earlier, ideally, TrigIt will be used as a bot running on a continuous integration server that sends code review with changes when a trigger holds; we do *not* expect developers to run TrigIt in the default build profile on code that is automatically deployed.

3.3 Implementation

We implemented TrigIt as a standalone Java library that can be used from the command line or integrated into a build system. TrigIt uses the ASM Java bytecode manipulation and analysis framework [11] to rewrite executable code, check correctness of encoding, and transform executable code. More precisely, TrigIt uses the visitor mechanism to build the model classes of the entire project, which are queried with the stream operations. TrigIt executes the action statements, one by one, which transform the underlying bytecode using ASM.

Additionally, we implemented a Maven plugin to simplify the integration of TrigIt into the build process, as Maven is still one of the most popular build systems for Java. We integrated the plugin in the Maven lifecycle after the compilation phase.

TrigIt provides various options, including "–debug" to show execution steps and store stripped files used to evaluate TrigIt methods, "–assume-true" to force the evaluation of triggers to true, and "–no-action" to tell that no action should be taken. "–assume-true" can be used to check the effect of executing the actions and "–no-action" can be used to check correctness of encoding and report what actions would be taken in the current build run.

We currently support basic checks of correctness of executable trigger-action comments. Specifically, TrigIt checks if a query expression and action statements refer to code elements that should exist. As an example, consider the following trigger:

```
TrigIt.getClass("C").getField("f").isPrivate()
```

If class `C` or field `f` does not exist, we report the incorrect encoding. These checks are important to detect modifications in code, likely due to software evolution, which invalidate executable comments and notify developers.

Finally, TrigIt can output a patch for the source code if the trigger evaluates to true and actions are executed. Our current approach for creating a patch utilizes the debug info to get the source code location of the patch, and aggregates the patches generated from each executed global action and local action. The patch also removes any TrigIt specification if its trigger evaluates to true, thus avoiding technical debt. We expect that a developer would inspect a patch and revise their code manually.

4 USER STUDY

This section describes our user study *to evaluate whether developers can quickly implement correct triggers and actions using TrigIt.*

4.1 Study Design

Initially, we asked each participant to read a tutorial for up to ten minutes. The tutorial provided a brief explanation of the TrigIt DSL with three examples. Next, we asked each participant to do three tasks; the tasks were chosen randomly from a set of executable trigger-action comments (from open-source projects) we encoded in the TrigIt DSL and are independent from the examples in the tutorial. Each task asked the participant to migrate an existing trigger-action comment to the TrigIt DSL. We did not ask the users to run TrigIt as that was outside the scope of our study; recall that our goal was to evaluate encoding of comments. We provided only a brief description of each task with the goal to clarify triggers and actions. This was necessary, because the participant, unfamiliar with the project, may not be able to infer the unspecified part of triggers and actions, and our goal was not to evaluate the understanding of code and comments but rather to evaluate the complexity of encoding the comments. Due to space limit, the detailed content of the tutorial and the descriptions are available on the accompanying web page [54]. We sent the tasks in the same order to each participant, and we used the first task as a warm-up task without telling this to the participants; the warm-up task was not considered in the evaluation. We provided a bash script to install tools, unpack projects that contain todo comments, and start the tasks. Our script sets up IntelliJ to provide a uniform environment with code completion.

We asked participants to track time spent on each task, to rate the level of confidence of their solutions for each task separately, and to send us their solutions. Then we asked them to rate the easiness of learning the TrigIt DSL (on a scale of 1—5, with 5 being the easiest). Finally, we asked them to tell us about their programming experience, experience with Java and experience with IntelliJ. We wrote scripts to process the responses, and we manually checked the correctness of each solution.

Participants. Our study was conducted in two batches. In the first batch, we sent the study to 15 people, including seven professionals (six working for large software companies and one researcher) and eight students (three undergraduate and five PhD). We received 12 valid responses in total. One (PhD student) participant did not follow the instructions. Two participants (one PhD student and one professional) had issues with running our bash scripts on OS X, specifically because of incompatible versions of sed, `find` and Java Development Kit between Linux and OS X. In the second batch, we sent the study to eight people, including two professionals (one working for large software company and one researcher) and six students (one undergraduate and five PhD). We received valid responses from all participants.

Table 2: Study Results Grouped By Tasks and Roles: Time Spent (Unit: Minute), Participants' Confidence (on a Scale of 1—5), and Correctness; Tri., Act. and Syn. are Numbers of Participants Who Got Correct Triggers, Actions and Syntax.

Grouped		Time		Confidence		Correctness		
By		Avg.	Med.	Avg.	Med.	Tri.	Act.	Syn.
Task	A	4.7	4.5	4.2	4.5	12/12	12/12	9/12
	B	4.3	5.0	4.4	4.5	12/12	12/12	12/12
	C	7.6	7.5	3.9	4.0	8/8	7/8	8/8
	D	7.2	6.5	4.2	4.0	7/8	8/8	8/8
Role	Prof.	5.0	5.0	4.5	5.0	16/16	15/16	15/16
	Stu.	6.1	5.5	4.0	4.0	23/24	24/24	22/24
	All	5.7	5.0	4.2	4.0	39/40	39/40	37/40

Among all 20 valid responses, the participants have on average 8.4 years (median: 6.0 years) of programming experience, and have moderate Java skills (average self-reported score 3.7 on a scale of 1—5); 12 participants have used IntelliJ before the study.

Tasks. We randomly chose tasks from the corpus of trigger-action comments mined from open-source projects that we previously encoded with the TrigIt DSL. The tasks (excluding the warm-up task) were:

- **TaskA** from `google/guava`: *"check more preconditions (as bufferSize >= chunkSize) if this is ever public"*. See Figure 1.
- **TaskB** from `apache/struts`: *"this is to keep backward compatibility, remove once when tooltipConfig is dropped"*. See Figure 8a.
- **TaskC** from `apache/ignite`: *"this comparison should be switched back to assertEquals when* `https://issues.apache.org/jira/browse/IGNITE-7692` *is fixed"*. See Figure 8b.
- **TaskD** from `jenkinsci/jenkins`: *" remove once Minimum supported Remoting version is 3.15 or above"*. See Figure 8c.

In the first batch we used TaskA and TaskB; in the second batch we used TaskC and TaskD. We confirmed that none of our participants contributed to the open-source projects used in the study.

4.2 Results

Table 2 summarizes the results, grouped by tasks and roles (professionals: Prof., students: Stu.). For each group, we show the average and median time in minutes to complete the task (Time), the average and median of participants' confidence on scale 1—5 (Confidence) and the number of participants that wrote the correct solution (Correctness). We access the correctness of different parts of the solution: *trigger* (Tri.), *action* (Act.), and *syntax* (Syn.) that we defined as correct locations for triggers and actions.

We can see that, on average, the participants took 5.7 minutes to migrate a trigger-action comment in an unfamiliar project; professionals took less time than students (average time: 5.0 minutes vs. 6.1 minutes). Most of the solutions were correct. There was one mistake on the trigger part and one mistake on the action part, both because of misunderstanding of the executable trigger-action comments. There were three mistakes on the syntax part of TaskA because the users put the local action inside the TrigIt method; we show a representative mistake in Figure 8d. Compared to the correct solution in Figure 1, the local action

```
// UIBean.java
public void evaluateParams() { ...
  // TODO: this is to keep backward compatibility, remove
  // once when tooltipConfig is dropped
  if (TrigIt.getCurrentClass().hasField("tooltipConfig")) {
    String jsTooltipEnabled = (String) getParameters()
                              .get("jsTooltipEnabled");
    if (jsTooltipEnabled != null)
        this.javascriptTooltip = jsTooltipEnabled;
  } ... }
```

(a) TaskB from the apache/struts project.

```
// IgniteCacheLockPartitionOnAffinityRunTest.java
private static int getPersonsCountSingleCache(final IgniteEx ignite,
  IgniteLogger log, final int orgId) throws Exception { ...
  // TODO this comparison should be switched back to assertEquals
  // when https://issues.apache.org/jira/browse/IGNITE-7692 is fixed.
  if (TrigIt.isClosed("https://issues.apache.org/jira/" +
     "browse/IGNITE-7692")) {
    assertEquals(partCnt, sqlCnt);
    assertEquals(partCnt, sqlFieldCnt);
  } else {
    if (partCnt != sqlFieldCnt)
      assertFalse("...", primaryPartition(ignite, orgId));
    if (partCnt != sqlCnt)
      assertFalse("...", primaryPartition(ignite, orgId));
  } ... }
```

(b) TaskC from the apache/ignite project.

```
// MasterToSlaveCallable.java
// TODO: remove once Minimum supported Remoting version is 3.15 or above
public Channel getChannelOrFail() throws ChannelClosedException
  { ... }

@TrigItMethod
void trigItMinRemotingVersion() throws Exception {
  if (((Version) TrigIt.getBuildConfigurations()
    .getProperty("remoting.minimum.supported.version"))
    .greaterEqualThan("3.15")) {
    TrigIt.getMethod(getChannelOrFail()).remove();
  } }
```

(c) TaskD from the jenkinsci/jenkins project.

```
// AbstractStreamingHasher.java
protected AbstractStreamingHasher(int chunkSize, int bufferSize) {
  // TODO(kevinb): check more preconditions
  // (as bufferSize >= chunkSize) if this is ever public
  checkArgument(bufferSize % chunkSize == 0); ... }

// SYNTAX INCORRECT
@TrigtItMethod
void trigItPreconditionCheck() {
  if (TrigIt.getMethod("<init>", int.class, int.class).isPublic())
    checkArgument(bufferSize >= chunkSize);
}
```

(d) An example of an incorrect encoding for TaskA.

Figure 8: Tasks used in the user study (TaskA is already shown in Figure 1) and one example of an incorrect solution to TaskA.

`checkArgument(bufferSize >= chunkSize)` was incorrectly put in the TrigIt method. After the study was done, we implemented checks in TrigIt to prevent such mistakes.

The participants claimed to be confident with their solutions (average score: 4.2); professionals are more confident than students (average score: 4.5 vs. 4.0).

Based on the results, we can say that participants obtained correct solutions with very little training. We consider these results even more encouraging because participants worked with unfamiliar projects. We believe that participants would perform better in a familiar environment and avoid mistakes due to the misunderstanding of comments. Finally, most participants consider TrigIt easy to learn (average score: 4.0).

5 CASE STUDIES

In this section, we describe our process of mining trigger-action comments in open-source projects and encoding them as executable trigger-action comments in TrigIt. As additional evaluation, we measure the overhead of TrigIt on the build process, as well as the number of tokens needed to encode executable trigger-action comments compared to the existing comments.

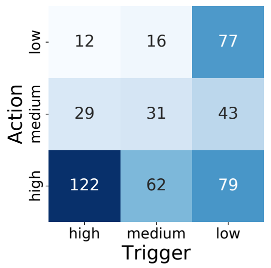
5.1 Projects and Comments

In the first step, we selected ten popular open-source projects to find trigger-action comments. Table 3 shows the list of projects (first column) and revisions used in our experiments (second column). We selected projects that differ in size, number of todo comments, and application domain. More importantly, we selected projects based on our prior experience with codebases. The last requirement was necessary to make the experiments feasible [55]; we wanted projects that we can build to ensure that we can run our tool after migrating the comments. In the second step, we extracted all todo comments from the selected projects. We searched for "TODO", which is the most common marker for todo comments [49]. Column 3 in Table 3 shows the number of todo comments for each project.

Table 3: Projects Used in our Case Studies, Number of Comments with "TODO" Marker (#TODO), Number of Trigger-Action Comments (#TAC), and Number of Other Types of Comments (#Other).

Project	Revision	#TODO	#TAC	#Other
apache/cayenne	9c07e18	379	9	370
apache/ignite	299f557	426	91	335
apache/struts	e2c2ea8	62	6	56
elastic/elasticsearch	850e9d7	436	66	370
google/closure-compiler	3d4f525	918	85	833
google/guava	ea66419	1298	96	1202
google/j2objc	e85caea	327	58	269
java-native-access/jna	c333527	93	2	91
jenkinsci/jenkins	043abd8	393	57	336
jenkinsci/gmaven	80d5f66	28	1	27
Avg.	N/A	436.0	47.1	388.9
Σ	N/A	4360	471	3889

In the third step, we manually inspected all 4360 todo comments and labeled each comment with "yes" (if the comment is a trigger-action comment) or "no" (if the comment is not a trigger-action comment). Columns 4 and 5 in Table 3 show for each project the number of labeled trigger-action comments and the number of other types of comments. The inspection was done by three authors of this paper together, and in addition to the comment itself, we inspected the context of the comment, i.e., surrounding source code, and potentially other files in the project. We discussed each comment until we reached unanimous agreement.



In the fourth step, we inspected 471 trigger-action comments that we annotated with "yes" in the previous step and assigned values to two more labels: trigger_specificity, and action_specificity. Specificity can take one of the following values: (1) "high", which means that we can understand the trigger/action and migration should be feasible, (2) "medium", which means that we mostly understand the trigger/action and migration could potentially be done, and (3) "low", which means that we cannot understand the trigger/action or migration is not feasible. The heatmap above shows the distribution of labels for 471 trigger-action comments. We illustrate the assignment of labels using several examples:

- *"this is to keep backward compatibility, remove once when tooltipConfig is dropped"* from `apache/struts`; trigger_specificity: "high", action_specificity: "high".
- *"When FieldAccess detection is supported, mark that class as reachable there, and remove the containsPublicField flag here"* from `google/j2objc`; trigger_specificity: "medium", action_specificity: "medium".
- *"Enable testing for unused fields when ElementUtil glitch is fixed"* from `google/j2objc`; trigger_specificity: "low", action_specificity: "medium".
- *"embedded Derby Mode... change to client-server once we figure it out"* from `apache/cayenne`; trigger_specificity: "low", action_specificity: "low".

Finally, we manually encoded 44 trigger-action comments as executable trigger-action comments in TrigIt, and used them as subjects in the following evaluations. We randomly selected these trigger-action comments from only those comments where trigger_specificity: "high".

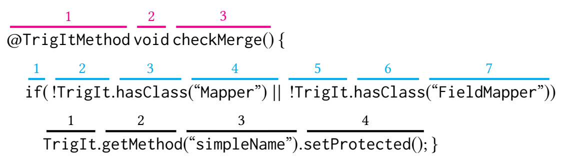
5.2 Build Overhead

We compute the build overhead to illustrate the cost of TrigIt analysis and code transformation. Recall that TrigIt runs after compilation and prior to program/test execution.

To compute the time overhead introduced in the build process, we measured the build time (e.g., `mvn test-compile` for Maven projects) for each project with and without TrigIt. TrigIt does *not* introduce any runtime overhead, so we do not run tests during the build. (Moreover, transformations performed by action statements may impact tests, i.e., new or different code may be executed, thus including test execution time might lead to misleading results.) If the build is run with TrigIt, we include all the phases, i.e., building the model of the project, evaluating the trigger, and executing the action (see Section 3). We run each configuration five times and compute the average time.

We ran experiments on an Intel i7-6700U CPU @ 3.40GHz with 16GB of RAM, running Ubuntu 18.04 LTS.

We find that the overhead varies between 1.91% and 26.75%; the average overhead is 8.85% and the median is 6.63%. We consider this to be acceptable overhead especially considering the sizes of the analyzed projects.



```
@TrigItMethod void checkMerge() {
  if(!TrigIt.hasClass("Mapper") || !TrigIt.hasClass("FieldMapper"))
    TrigIt.getMethod("simpleName").setProtected(); }
```

Figure 9: Example of token counting; we show number of tokens in the boilerplate code (first line), trigger (second line), and action (third line).

5.3 Complexity of the TrigIt DSL

We *estimate* the complexity of writing executable trigger-action comments with the number of tokens needed to encode triggers and actions. Concretely, we count three parts for each executable trigger-action comment: (1) trigger, which is the number of tokens to encode the trigger, (2) action, which is the number of tokens to encode the action, and (3) boilerplate code, which is the number of tokens to satisfy the TrigIt DSL syntax, e.g., method signature. Figure 9 illustrates, using an earlier example, the way we count the tokens. We compare the complexity of executable trigger-action comments with their informal counterparts written in natural language.

We found that the average number of tokens in the original comments was 12.2 (median: 12.0, min: 6, max: 26), while the total number of tokens in the encoded comments (trigger + action + boilerplate code) was, on average, 18.0 (median: 16.5, min: 11, max: 36). Additionally, the number of tokens in the triggers was, on average, 7.3 (median: 6.0, min: 4, max: 19); the number of tokens in the actions was, on average, 7.4 (median: 5.0, min: 2, max: 27); the average number of tokens in the boilerplate code was 3.3 (median: 3.0, min: 3, max: 7).

The results show that even if we count all the tokens in executable trigger-action comments, the increase in the number of tokens compared to the informal text is only 58.74%. If we take into account only the tokens in triggers and actions, executable trigger-action comments are in some cases even shorter than the original comments. Additional benefits of the executable trigger-action comments is that developers can utilize the features of IDEs, such as auto-completion and generation of method signatures, which do not work for comments written in natural language.

6 ANECDOTAL EXPERIENCE

We briefly report on our experience on using TrigIt and interacting with open-source developers to understand their comments. To encode each comment, we read the comment and encoded what we believe were valid triggers and actions. In several cases, we observed that TrigIt always executes the actions, and we thought that there was a bug in our tool. However, by looking at those triggers manually, we found that the triggers are satisfied but the actions have not been executed. Table 4 shows the time when the comment was included in the code repository, time when the trigger is satisfied, and when the action is executed. We only show comments where the trigger is satisfied. Note that `FreemarkerResultMockedTest` is a special case, because the developer forgot to remove the todo comment even after the action was taken. We found this todo comment interesting and used an older revision of the project in our evaluation. We reported to the corresponding

Table 4: Timeline of the Comments: Added (Column 3-4), Trigger Satisfied (Column 5-6), Action Executed (Column 7-8).

Project	Class	First Added		Trigger Satisfied		Action Executed	
		Revision	Date	Revision	Date	Revision	Date
apache/cayenne	DeduplicationVisitor	`39b70d1`	2016-10-02	`b332610`	2017-08-18	After we reported	
apache/struts	FreemarkerResultMockedTest	`0f2c049`	2012-11-22	`25cdfd6`	2015-05-28	`a5812bf`	2015-10-06
google/closure-compiler	DependencyInfo	`fc465c1`	2016-04-25	`62ba0ab`	2017-10-11	N/A	N/A
google/guava	AbstractFuture	`0b76074`	2014-11-25	`86fb700`	2016-11-04	N/A	N/A
google/guava	ClassPath	`896c51a`	2017-01-12	`9ebd95a`	2018-02-20	N/A	N/A
google/j2objc	GeneratedExecutableElement	`6eac122`	2016-12-14	`bc5dbad`	2017-08-31	N/A	N/A

developers five comments that have satisfied triggers but not executed actions. We got responses from all developers just a few hours after we submitted the reports. A developer of `apache/cayenne` immediately performed the action and sent a note: "Thank you for the reminder!" A developer of `google/j2objc` confirmed that the trigger is satisfied and said: "it's time to cleanup the TODOs – any volunteers? :-)". Developers of other comments explained that the comments were not specific enough:

- *"Unfortunately Java 8 still causes some issues with some Google-internal infrastructure. Java 8 is allowed in tests but not in the main part of the code (yet)."*
- *"The comment should say something like 'when jdk8 is available to all flavors of Guava.' We currently maintain a backport that targets JDK7 and older versions of Android, and we try to keep the backport and mainline mostly in sync. That's not to say that we couldn't do this, but we'd have to weigh the benefits against the cost of diverging the two."*

Some of the responses confirmed the impression that we had while reading todo comments: knowing the details of the project is likely necessary to do valid migration of comments to TrigIt. However, this is not surprising considering that todo comments are written to communicate among developers of the project.

7 DISCUSSION

Other observations related to comments. We encountered a large number of interesting cases while analyzing the comments. We describe only one case here due to the limited space. We observed several examples with a trigger that depend on a test case, e.g., *"delete this variable and corresponding if statement when jdk fixed java.text.NumberFormat.format's behavior with Float"*. We plan to explore how to encode triggers based on test execution results.

Naïve alternatives. Writing tests or throwing exceptions can be used to partially encode trigger-action comments, e.g., a test can fail or an exception can be thrown when Java version is 7. TrigIt overshadows these naïve approaches in three ways: (1) TrigIt can perform actions that modify code; (2) TrigIt provides an API for querying the codebase, build scripts, bug tracking systems, and system clock time; and (3) unlike TrigIt specifications, exceptions would remain in compiled code, which could lead to unexpected behaviors once software is deployed.

Future work. (1) We qualitatively evaluated TrigIt's benefit via discussion with developers in industry and the user study; we envision a systematic cost-benefit analysis once TrigIt is adopted in real-world developing process. (2) The current implementation does not consider dependencies (and conflicts) between executable trigger-action comments, although we have not observed any yet. For example a trigger from one comment may become true when the action of another comment is executed. To support the correct order of execution, we will need to maintain dependencies between comments. (3) With recent advances of general-purpose code synthesis from natural language [2, 3, 12, 60], it is worth exploring training a semantic parser to automatically map natural language comments to TrigIt specifications, which will remove the burden from developers. Our initial work in this direction focused on identifying trigger-action comments in a given repository [36].

8 THREATS TO VALIDITY

External. We extracted only comments containing "TODO" markers, however, developers use other markers, including "FIXME", "XXX", "HACK", etc. Our decision was based on prior work that showed that "TODO" is the most common marker [49].

The projects that we used in the evaluation may not be representative of all open-source projects. To mitigate this threat, we used popular open-source projects that are actively maintained.

TrigIt supports only projects written in the Java programming language. We chose Java because it is one of the most popular languages, and prior work on analyzing (todo) comments showed the need for automating comment maintenance (see Section 9). However, the idea behind TrigIt is broadly applicable.

We encountered challenges in recruiting a large number of experienced participants in our user study. However, we consider 20 participants, including eight professionals, to be a sufficiently large group for our kind of study.

Internal. Our scripts for mining repositories and TrigIt code may contain bugs. We used scripts already utilized in prior work, and we manually inspected the results of some of those scripts.

Construct. The focus of TrigIt is on triggers and actions related to the content of compiled code, build files, issue tracking systems, and system clock. Many comments belong to these categories. Support for queries that check data available only in the source code, e.g., compile time annotations, are left for future work.

Time taken to perform the study was self-reported by the participants. Although we were considering to capture the screen, participants rejected this option. As reported times are similar with the time it took us to write those comments, we have no reason to believe that anybody reported incorrect numbers.

9 RELATED WORK

Comment analysis and automation. Ying et al. [61] were among the first to identify the importance and frequency of todo comments. They analyzed two groups of comments that are a subset of trigger-action comments: "communication: self-communication" and "future task: once the library is available...". Empirical studies done by Storey et al. [49] and Haouari et al. [22] confirmed that todo comments are ubiquitous and may lead to maintenance issues. Sridhara [46] developed a rule-based system for identifying out-of-date todo comments. Nie et al. [35] proposed several techniques for comment and program analysis to support todo comments as software evolves. Pascarella and Bacchelli [40] performed manual classification of comments and showed that machine-learning has potential to automate the classification. Innobuilt Software developed `imdone` [31], an online tool that extracts and tracks todo comments by creating and updating issues (e.g., on GitHub or JIRA). TrigIt is motivated by prior work on identifying and analyzing todo comments, and our main goal is to simplify maintenance of comments and code.

Prior work studied detecting and eliminating the inconsistency between code and comments. For example, Fluri et al. [15] studied co-evolution of code and comment; Tan et al. developed iComment [51] and tComment [52] to detect code/comment inconsistencies; Ibrahim et al. [25] found that rare inconsistent updates lead to bugs in future revisions of software; Svensson [50] developed a program for manual comment consistency checks; Ratol and Robillard [44] and Zhou et al. [63] looked at the identifiers in API or comments that may become inconsistent as the result of a code refactoring (e.g., renaming). `todo_or_die` [45] is a tool for keeping todo comments up-to-date by specifying a date for each comment, and breaking the execution upon outdated todo comments. TrigIt is the first solution towards avoiding inconsistencies between code and trigger-action comments, by automatically maintaining code repository and executing actions when associated triggers hold. TrigIt also removes the need for detecting identifiers in comments as the trigger-action comments are encoded as executable Java code, which would be refactored together with other code.

Work on self-admitted technical debt [8, 24, 32, 42, 62] (SATD) identifies comments that document temporary code fixes. Unlike work on identifying SATD, TrigIt could be used to clean up the codebase when the trigger condition is satisfied.

Work on generating comments from code [20, 23, 27, 33, 34, 47, 48] , generating code / specifications from comments [9], and both directions [41], are other approaches to keep a repository consistent. TrigIt provides a way to write executable comments that automatically and consistently update both code and comments as the codebase evolves.

Code query languages. JQuery [28, 56] and CodeQuest [21] are source code querying tools; the former uses a logic programming language, while the latter uses Datalog. Recently, Urma and Mycroft [57] proposed source-code queries with graph databases. While prior work mostly targeted program comprehension, TrigIt targets encoding of trigger-action comments with a language embedded in Java. Another difference is that TrigIt supports querying various other artifacts, e.g., build configuration files.

Ozdemir et al. developed a tool, built on their prior work CodeAware [1], for monitoring code repositories and notifying developers if some code metrics (e.g., complexity) change [39]; TrigIt is for encoding executable comments, querying code repositories, and transforming the program.

The Reflection API [5] can be used to query code, but other types of queries supported in TrigIt cannot be written using this API.

Program transformations. Actions available in TrigIt are closely related to behavior-preserving transformations, i.e., refactorings [17, 37, 38, 53]. Most relevant work is that on scripting refactorings [29, 58, 59], i.e., providing simple building blocks that can be composed in sophisticated transformations. One of the key differences is that TrigIt actions may not be behavior-preserving, e.g., using new API calls or removing a statement. If actions that we discover in the future would require complex code transformations offered by existing refactoring engines, it would be worth integrating our actions with those engines.

TrigIt methods and their encoding follow similar patterns as the AOP condition-action patterns [14]. However, in TrigIt the "aspect" code lives in the same place as the regular code and global actions can change the internal and external API.

IFTTT (if-this-then-that) recipe synthesis. Researchers have studied synthesizing IFTTT recipes from natural language [10, 12, 30, 43, 60]. IFTTT recipes are short scripts of trigger-action pairs in daily life domains such as smart home, personal well-being and social networking, shared by users on websites such as IFTTT.com. Frequently a recipe is accompanied by a short natural language description. In IFTTT recipes, triggers and actions are functions from APIs and services (e.g., Instagram). In our work, triggers and actions are drawn from unstructured todo comments during software development. Instead of translating from natural language to an already developed target programming language, our goal is to develop the target programming language.

10 CONCLUSION

We presented the first approach, dubbed TrigIt, to encode trigger-action comments as executable statements. A developer can use a Java-like language to encode triggers as query statements over artifacts related to the code repository and actions as code transformation steps. TrigIt integrates into the build process and interposes between the program compilation and execution. We migrated 44 trigger-action comments from several large open-source projects. Evaluation of TrigIt, via a user study, showed that users find TrigIt easy to learn and use. Additionally, we showed that TrigIt introduces negligible overhead in the build process and the number of tokens needed to encode the comments differs only slightly from the original comments written in natural language. Although TrigIt could be extended in several ways, we believe that timely code updates enabled by TrigIt can already have positive impact on code comprehension and maintenance.

ACKNOWLEDGMENTS

We thank Ahmet Celik, Nima Dini, Lamyaa Eloussi, Alex Gyori, Vilas Jagannath, Yun Young Lee, Yu Lin, Darko Marinov, Aleksandar Milicevic, Stas Negara, Karl Palmskog, Sheena Panthaplackel, Marinela Parovic, Jesse Piascik, Rohan Sharma, Kaiyuan Wang, Tifany Yung, Zhiqiang Zang, Chenguang Zhu, and the anonymous reviewers for their feedback on this work. We also thank all the user study participants. This work was partially supported by the US National Science Foundation under Grant Nos. CCF-1652517 and CCF-1704790.

REFERENCES

[1] Rui Abreu, Hakan Erdogmus, and Alexandre Perez. 2015. CodeAware: Sensor-based fine-grained monitoring and management of software artifacts. In *International Conference on Software Engineering*, Vol. 2. 551–554.

[2] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *Comput. Surveys* 51, 4 (2018), 81.

[3] Miltos Allamanis, Daniel Tarlow, Andrew Gordon, and Yi Wei. 2015. Bimodal modelling of source code and natural language. In *International Conference on Machine Learning*. 2123–2132.

[4] Oracle and/or its affiliates. 2019. Chapter 5. Loading, Linking, and Initializing. https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-5.html.

[5] Oracle and/or its affiliates. 2019. The Reflection API. https://docs.oracle.com/javase/tutorial/reflect/.

[6] Oracle and/or its affiliates. 2019. Stream (Java Platform SE 8). https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html.

[7] Apache. 2019. Apache Struts. https://github.com/apache/struts.

[8] Gabriele Bavota and Barbara Russo. 2016. A large-scale empirical study on self-admitted technical debt. In *International Working Conference on Mining Software Repositories*. 315–326.

[9] Arianna Blasi, Alberto Goffi, Konstantin Kuznetsov, Alessandra Gorla, Michael D. Ernst, Mauro Pezzè, and Sergio Delgado Castellanos. 2018. Translating code comments to procedure specifications. In *International Symposium on Software Testing and Analysis*. 242–253.

[10] Shobhit Chaurasia and Raymond J. Mooney. 2017. Dialog for language to code. In *International Joint Conference on Natural Language Processing*. 175–180.

[11] OW2 Consortium. 2018. ASM. http://asm.ow2.io.

[12] Li Dong and Mirella Lapata. 2016. Language to logical form with neural attention. In *Annual Meeting of the Association for Computational Linguistics*. 33–43.

[13] Elastic. 2019. Elastic Elasticsearch. https://github.com/elastic/elasticsearch.

[14] Robert E. Filman and Daniel P. Friedman. 2000. *Aspect-oriented programming is quantification and obliviousness.* Technical Report. Research Institute for Advanced Computer Science.

[15] Beat Fluri, Michael Wursch, and Harald C. Gall. 2007. Do code and comments co-evolve? On the relation between source code and comment changes. In *Working Conference on Reverse Engineering*. 70–79.

[16] Eclipse Foundation. 2019. Eclipse Java development tools (JDT). https://www.eclipse.org/jdt.

[17] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. 1999. *Refactoring: Improving the design of existing code.*

[18] Apache Gobblin. 2019. Apache Gobblin. https://github.com/apache/incubator-gobblin.

[19] Google. 2019. Google Guava. https://github.com/google/guava.

[20] Jhe-Jyun Guo, Nien-Lin Hsueh, Wen-Tin Lee, and Shi-Chuen Hwang. 2014. Improving software maintenance for pattern-based software development: A comment refactoring approach. In *International Conference on Trustworthy Systems and their Applications*. 75–79.

[21] Elnar Hajiyev, Mathieu Verbaere, and Oege de Moor. 2006. CodeQuest: Scalable source code queries with Datalog. In *European Conference on Object-Oriented Programming*. 2–27.

[22] Dorsaf Haouari, Houari Sahraoui, and Philippe Langlais. 2011. How good is your comment? A study of comments in Java programs. In *International Symposium on Empirical Software Engineering and Measurement*. 137–146.

[23] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *International Conference on Program Comprehension*. 200–210.

[24] Qiao Huang, Emad Shihab, Xin Xia, David Lo, and Shanping Li. 2018. Identifying self-admitted technical debt in open source projects using text mining. *Empirical Software Engineering* 23, 1 (2018), 418–451.

[25] Walid M. Ibrahim, Nicolas Bettenburg, Bram Adams, and Ahmed E. Hassan. 2012. On the relationship between comment update practices and software bugs. *Journal of Systems and Software* 85, 10 (2012), 2293–2304.

[26] Apache Incubator-Wave. 2019. Apache Incubator-Wave. https://github.com/apache/incubator-wave.

[27] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing source code using a neural attention model. In *Annual Meeting of the Association for Computational Linguistics*. 2073–2083.

[28] Doug Janzen and Kris De Volder. 2003. Navigating and querying code without getting lost. In *International Conference on Aspect-Oriented Software Development*. 178–187.

[29] Huiqing Li and Simon Thompson. 2012. A domain-specific language for scripting refactorings in Erlang. In *Fundamental Approaches to Software Engineering*. 501–515.

[30] Chang Liu, Xinyun Chen, Eui Chul Shin, Mingcheng Chen, and Dawn Song. 2016. Latent attention for if-then program synthesis. In *Advances in Neural Information Processing Systems*. 4574–4582.

[31] Innobuilt Software LLC. 2019. All your TODO comments in one place. https://imdone.io/.

[32] Everton da S Maldonado, Rabe Abdalkareem, Emad Shihab, and Alexander Serebrenik. 2017. An empirical study on the removal of self-admitted technical debt. In *International Conference on Software Maintenance and Evolution*. 238–248.

[33] Laura Moreno, Jairo Aponte, Giriprasad Sridhara, Andrian Marcus, Lori Pollock, and K Vijay-Shanker. 2013. Automatic generation of natural language summaries for Java classes. In *International Conference on Program Comprehension*. 23–32.

[34] Dana Movshovitz-Attias and William W Cohen. 2013. Natural language models for predicting programming comments. In *Annual Meeting of the Association for Computational Linguistics*. 35–40.

[35] Pengyu Nie, Junyi Jessy Li, Sarfraz Khurshid, Raymond Mooney, and Milos Gligoric. 2018. Natural language processing and program analysis for supporting todo comments as software evolves. In *Workshops of the the AAAI Conference on Artificial Intelligence*. 775–778.

[36] Pengyu Nie, Rishabh Rai, Junyi Jessy Li, Sarfraz Khurshid, Raymond J. Mooney, and Milos Gligoric. 2018. Executable trigger-action comments. *CoRR* abs/1808.01729 (2018).

[37] William F. Opdyke. 1992. *Refactoring object-oriented frameworks.* Ph.D. Dissertation. University of Illinois at Urbana-Champaign.

[38] William F. Opdyke and Ralph E. Johnson. 1990. Refactoring: An aid in designing application frameworks and evolving object-oriented systems. In *Symposium on Object-Oriented Programming Emphasizing Practical Applications*. 145–161.

[39] Alim Ozdemir, Ayse Tosun, Hakan Erdogmus, and Rui Abreu. 2018. Lightweight source code monitoring with Triggr. In *Automated Software Engineering, Tool Demonstrations*. 864–867.

[40] Luca Pascarella and Alberto Bacchelli. 2017. Classifying code comments in Java open-source software systems. In *International Working Conference on Mining Software Repositories*. 227–237.

[41] Hung Phan, Hoan Anh Nguyen, Tien N Nguyen, and Hridesh Rajan. 2017. Statistical learning for inference between implementations and documentation. In *International Conference on Software Engineering: New Ideas and Emerging Technologies Results Track*. 27–30.

[42] Aniket Potdar and Emad Shihab. 2014. An exploratory study on self-admitted technical debt. In *International Conference on Software Maintenance and Evolution*. 91–100.

[43] Chris Quirk, Raymond J Mooney, and Michel Galley. 2015. Language to Code: Learning semantic parsers for If-This-Then-That recipes.. In *Annual Meeting of the Association for Computational Linguistics*. 878–888.

[44] Inderjot Kaur Ratol and Martin P. Robillard. 2017. Detecting fragile comments. In *Automated Software Engineering*. 112–122.

[45] Justin Searls. 2019. todo_or_die. https://github.com/searls/todo_or_die.

[46] Giriprasad Sridhara. 2016. Automatically detecting the up-to-date status of ToDo comments in Java programs. In *India Software Engineering Conference*. 16–25.

[47] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K Vijay-Shanker. 2010. Towards automatically generating summary comments for Java methods. In *Automated Software Engineering*. 43–52.

[48] Giriprasad Sridhara, Lori Pollock, and K Vijay-Shanker. 2011. Generating parameter comments and integrating with method summaries. In *International Conference on Program Comprehension*. 71–80.

[49] Margaret-Anne Storey, Jody Ryall, R. Ian Bull, Del Myers, and Janice Singer. 2008. TODO or to bug. In *International Conference on Software Engineering*. 251–260.

[50] Adam Svensson. 2015. *Reducing outdated and inconsistent code comments during software development: The comment validator program.* Master's thesis. Uppsala University, Information Systems.

[51] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. 2007. /*iComment: bugs or bad comments?*/. In *Symposium on Operating Systems Principles*. 145–158.

[52] Shin Hwei Tan, Darko Marinov, Lin Tan, and Gary T. Leavens. 2012. @tComment: Testing Javadoc comments to detect comment-code inconsistencies. In *International Conference on Software Testing, Verification, and Validation*. 260–269.

[53] Lance Tokuda and Don Batory. 1999. Evolving object-oriented designs with refactorings. In *Automated Software Engineering*. 174–181.

[54] TrigIt. 2019. TrigIt web page. http://cozy.ece.utexas.edu/trigit.

[55] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. 2017. There and back again: Can you compile that snapshot? *Journal of Software: Evolution and Process* (2017), e1838.

[56] Raoul-Gabriel Urma and Alan Mycroft. 2012. Programming language evolution via source code query languages. In *Workshop on Evaluation and Usability of Programming Languages and Tools*. 35–38.

[57] Raoul-Gabriel Urma and Alan Mycroft. 2015. Source-code queries with graph databases—with application to programming language usage and evolution. *Science of Computer Programming* 97, P1 (2015), 127–134.

[58] Mohsen Vakilian, Nicholas Chen, Roshanak Zilouchian Moghaddam, Stas Negara, and Ralph E. Johnson. 2013. A compositional paradigm of automating refactorings. In *European Conference on Object-Oriented Programming*. Berlin, Heidelberg, 527–551.

[59] Mathieu Verbaere, Ran Ettinger, and Oege de Moor. 2006. JunGL: A scripting language for refactoring. In *International Conference on Software Engineering*. 172–181.

[60] Pengcheng Yin and Graham Neubig. 2017. A syntactic neural model for general-purpose code generation. In *Annual Meeting of the Association for Computational Linguistics*. 440–450.

[61] Annie T. T. Ying, James L. Wright, and Steven Abrams. 2005. Source code that talks: An exploration of Eclipse task comments and their implication to repository mining. In *International Working Conference on Mining Software Repositories*. 1–5.

[62] Fiorella Zampetti, Cedric Noiseux, Giuliano Antoniol, Foutse Khomh, and Massimiliano Di Penta. 2017. Recommending when design technical debt should be self-admitted. In *International Conference on Software Maintenance and Evolution*. 216–226.

[63] Yu Zhou, Ruihang Gu, Taolue Chen, Zhiqiu Huang, Sebastiano Panichella, and Harald Gall. 2017. Analyzing APIs documentation and code to detect directive defects. In *International Conference on Software Engineering*. 27–37.