SWIRL: Interactive Sensemaking of Tool-Generated Warnings through Customized Summaries

Burak Yetiştiren
Computer Science
UCLA
Los Angeles, CA, USA
burak@cs.ucla.edu

Hong Jin Kang
The University of Sydney
Sydney, Australia
hongjin.kang@sydney.edu.au

Miryung Kim
Computer Science
UCLA
Los Angeles, CA, USA
miryung@cs.ucla.edu

***Abstract*—Programmers using bug-finding tools often review their reported warnings one by one. Based on the insight that identifying recurring themes and relationships can enhance the cognitive process of searching for representations of a given problem space (i.e., *sensemaking*), we propose** SWIRL**, which supports interpreting tool-generated warnings through interactive, customized summarization. With active feedback,** SWIRL **derives summary rules for grouping of related warnings on the fly. As users mark warnings as interesting or uninteresting,** SWIRL**'s rule inference algorithm surfaces common characteristics, highlighting structural similarities in containment, subtyping, invoked methods, accessed fields, and expressions.**

We demonstrate SWIRL **on real-world warnings generated from Infer and SpotBugs on two mature Java projects. In a within-subject user study, our participants articulated root causes for similar uninteresting warnings with more confidence when using** SWIRL**, compared to the baseline that lists individual warnings without customized summary rules. Among participants, we observed significant individual variation in desired grouping, reinforcing the need for individualized sensemaking. The simulation we conducted shows that** SWIRL**'s rule-level feedback expedites sensemaking, requiring only 11.8 interactions on average to align all inferred rules with a simulated user's labels when combined with instance-level feedback, compared to 17.8 interactions when using instance-level feedback alone. Our evaluation suggests that** SWIRL**'s active learning-based summarization can enhance the sensemaking process of tool-generated warnings.**

***Index Terms*—Interactive warning inspection, Inductive logic programming, Sensemaking, Program Analysis**

I. Introduction

Bug-finding tools, such as Findbugs [1] and Infer [2], have been successful in detecting issues. However, they also produce numerous uninteresting warnings that the developers do not want to investigate further [3], [4], [5], [6]. Users often struggle to make sense of these, or easily distinguish them from the interesting warnings and this difficulty has caused reluctance to adopt these bug-finding tools into a developer workflow [3], [4], [7]. In our own analysis of the Alibaba Nacos open-source project [8], we find that 43% of the 58 `Null Pointer Dereference` warnings generated by Infer are never fixed, indicating that they are probably ignored by developers[1]. Evidently, the burden of sifting through a large number of tool-generated warnings has impeded the effective adoption of bug finding tools [3], [4], [5], [9]. Existing research has focused on filtering or reprioritizing warnings [10] but overlooked the problem of *sensemaking*, defined as the essential task of searching for a representation or representations of the problem space or the underlying structure of a given set of information. In our use case scenario, we define sensemaking as *making sense of related uninteresting (or interesting) warnings and identifying their common characteristics out of a pool of all warnings generated by a given bug-finding tool*.

Unlike approaches that ask "Is this single warning a false positive or a true bug?", sensemaking helps developers construct mental models for understanding the code and warnings reported on them. Inspired by the variational theory of learning [11], where a user should form accurate conceptualizations by discerning variations of a phenomenon, we introduce SWIRL (**S**ensemaking **W**arnings with **I**nteractive **R**eview and **L**earning). SWIRL surfaces the commonalities and differences between warnings upon interactive feedback from a user. SWIRL empowers developers to create customized flexible groups of warnings by showing a customized scaffolding constructed with user feedback to examine related warnings together.

SWIRL's interface is shown in Figure 1. Instead of viewing individual warnings one by one, using SWIRL, a user can formulate customized summaries of warnings by providing interactive feedback. This offers configurability, such that the summary rules are learned from user's labeling and code statement-level feedback on individual warnings. These summary rules reveal common characteristics based on code locations, type hierarchies, and similar code statements. Therefore, the user can easily make sense of similarities and dissimilarities between different inferred groups.

SWIRL provides the following capabilities to improve sensemaking:

Feedback-driven Alignment. SWIRL infers/re-infers summary rules, as a user iteratively provides more feedback. This introduces a feedback loop. In this process, the warnings are grouped by rules, which is by no means filtering out any warnings, but assigning them to groups when applicable (when a warning matches an inferred group).

[1]based on our analysis in https://github.com/UCLA-SEAL/SWIRL/tree/master/code/neverfixed_analysis

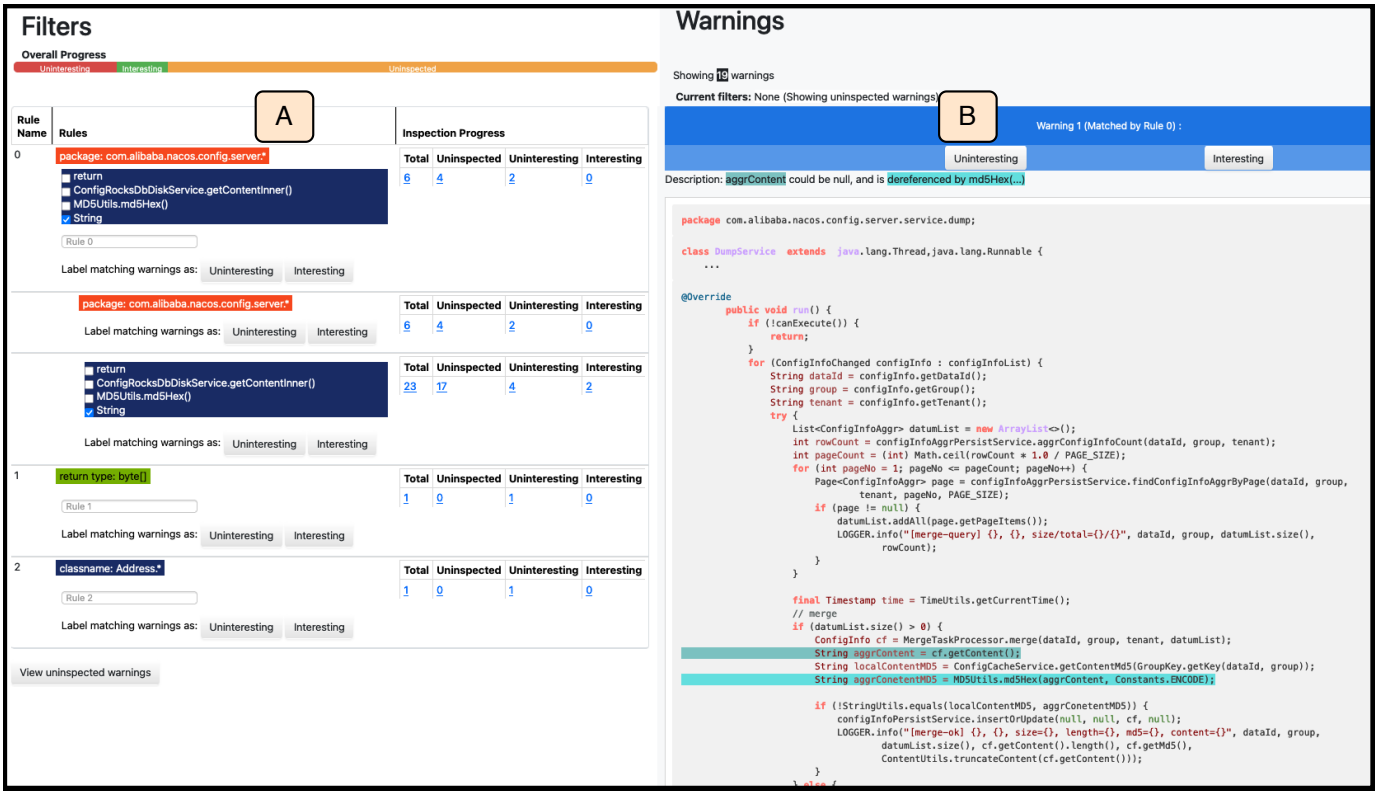


Fig. 1: SWIRL guides users in *sensemaking* of tool-generated warnings with inductive summary rules. Users can provide explicit feedback via (A) checkmarking code expressions or (B) highlighting code snippets. SWIRL then refines the summary rules interactively.

SWIRL employs inductive logic programming (ILP) to derive rules over containment, typing relations, and similar code expressions. These relationships capture the implicated code's scope, type hierarchy, and the API signature of a method where a warning is reported (e.g., the return type of the method, or the class fields that the method uses). This is useful since uninteresting warnings are often correlated by locality and organization of the implicated code [12].

Active Learning. Active learning traditionally refers to learning settings in which the learner exercises some control over the data from which it learns, typically by selecting unlabeled instances and querying an oracle, such as a human annotator, for their labels [13], [14]. In our work, we use the term active learning in a broader, user-directed sense. Rather than having the learner select the instances to be labeled, we allow users to choose warning instances and provide labels based on their own sensemaking needs (interactive rule induction). The tool then incorporates these labels to iteratively induce and refine personalized grouping rules. Thus, our approach retains the iterative label–learn–update cycle of active learning, but places control over instance selection with the user rather than the learning algorithm.

Visualization. With this focus on interactive sensemaking, SWIRL's user interface is different from a typical user-interface of bug-finding tools such as SpotBugs (as shown in Figure 2). Instead of displaying warnings in a tree view according to predefined filters or organizing them based on file locations, SWIRL organizes warnings with custom summaries created with interactive feedback.

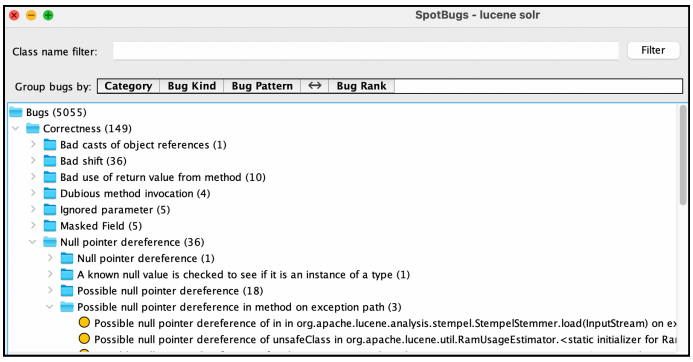


Fig. 2: User interface of SpotBugs, a typical bug finding tool, which groups warnings by analysis kind or bug pattern. The lack of configurability for summarizing related warnings forces users to inspect warnings one-by-one.

To evaluate SWIRL, we conducted a user study and a simulation experiment. First, we carried out a within-subject user study with 14 participants. The participants inspected null pointer dereference warnings reported by Infer and Spotbugs on two large mature projects (i.e., real warnings from real

systems). They were asked to articulate commonalities between uninteresting warnings. We built a baseline for inspecting warnings one-by-one, similar to conventional tree-based organization bundled with Infer [15], which reflects current developer practice [3], [4]. Rather than measuring how many warnings participants inspected, we evaluated how well they could articulate commonalities among related warnings.

A total of 55 different rules were formulated by 14 participants, with an average of 3.9 rules per participant. When using SWIRL, the participants reported lower levels of mental demand (4.8 vs. 5.4 on a 7-point scale) and reported greater confidence (3.4 vs. 2.6 on a 5-point scale), compared to the baseline. Using SWIRL, participants were able to better articulate commonalities among uninteresting warnings. They also wrote longer, more detailed descriptions about the commonalities. We found significant individuality among the summary rules constructed by individual participants, indicating that developers would desire greater configurability during their sensemaking process. Moreover, the significant individuality among rules confirms that warning inspection is an inherently subjective task, showing the benefit of personalized groups as opposed to universal pre-defined rules.

As the second evaluation method, we conducted simulation experiments to quantify the benefit of rule-based feedback and summarization. We observed, when a user provides rule-level feedback as opposed to instance-level feedback, rules inferred by SWIRL become aligned faster with interactive feedback. On average, for the warnings found in Alibaba Nacos, within 12 iterations, the rule alignment becomes over 80%, and within 18 iterations, rule alignment becomes 100%.

In summary, this paper makes the following contributions:

1) We design a personalized, interactive sensemaking workflow for tool-generated warnings, in which ILP-derived summary rules are iteratively refined through user feedback to highlight common characteristics by code location, type hierarchy, API signatures, or recurring expressions.
2) In a within-subject user study, participants using SWIRL were better able to articulate commonalities among uninteresting warnings and reported greater confidence in their inspections.
3) A simulation study shows that summary rules align more quickly with rule-level feedback than with instance-level feedback.

The rest of this paper is organized as follows. Section II describes a usage scenario of SWIRL. Section III introduces SWIRL's active learning approach. Section IV presents the design of our user study. Section V presents our simulation experiment design and its results. Section VI discusses our study's implications and possible threats to validity. Section VII presents related work. Finally, we draw the conclusions of our work in Section VIII.

II. Motivating Example

Overview. To motivate our approach, we introduce a running example from Infer's null-pointer analysis (Fig. 3). The yellow highlights mark the null pointer dereference warnings reported by the analyzer. Although these warnings look similar, their actionability differs: (a)–(c) are less interesting warnings, whereas (d) reflects a genuine defect. We use this example to first show why one-by-one inspection may hinder sensemaking, and then to demonstrate how SWIRL turns a few pieces of user feedback into summary rules that group similar uninteresting warnings.

A. Lack of sensemaking support in existing tools

Suppose that Alice is given the task of inspecting null pointer dereference warnings generated by a static analyzer (Infer). Upon running it, Alice is overwhelmed by the large number of tool-generated warnings. After inspecting several of them, Alice is frustrated and cannot easily make sense of which warnings should be acted upon. Figure 3a shows an example null pointer dereference warning reported by Infer.

*a) **Difficulty of sensemaking when inspecting warnings one-by-one**:* Reviewing each warning one-by-one, Alice determines if each warning is worthwhile to address or should be ignored. After inspecting a few, Alice notices that several uninteresting warnings share similar characteristics, for example, being located in the same package or invoking similar API calls. This process remains burdensome and frustrating, as Alice cannot easily check and articulate the underlying characteristics of similar recurring warnings.

*b) **Difficulty of sensemaking when using predetermined filters**:* Alice may decide to determine a set of rules to triage groups of related tool-generated warnings. She observes that many warnings from the package `com.alibaba.nacos.persistence` are uninteresting for the same reason (e.g., a field set with the value fetched from a database is not `null`). At this point, she manually writes a script to identify warnings in the package `com.alibaba.nacos.persistence`. Consequently, she observes that several warnings related to a field set through serialization tend to be less interesting to her; she hypothesizes that these warnings tend to be located in package `com.alibaba.nacos.config.server`. At the same time, she realizes that such warnings should be refined by subtyping relations, since they are located in classes implementing database-related interfaces. To mitigate this, she has to revise the existing script or write another one. Alice realizes that it is difficult to pre-define a set of rules, since the sensemaking process requires interactive navigation, search, and summarization of similar warnings.

B. SWIRL's interactive sensemaking

Alice then gives SWIRL a try. After labeling a few uninteresting warnings, she notices that the tool automatically proposes summary rules to group similar ones. These rules draw on containment relations, type hierarchies, API signatures of implicated methods, and recurring expressions. For instance, a rule may capture all warnings in the package `com.alibaba.nacos.config.server`. Below, we discuss two kinds of sensemaking processes in more detail.

```
Throwable ex = null;
int maxRetry = getProperty(
    ADDRESS_SERVER_RETRY_PROPERTY, Integer.class
    , DEFAULT_SERVER_RETRY_TIME);
for (int i = 0; i < maxRetry; i++) {
    try {
        ...
        success = true;
        break;
    } catch (Throwable e) {
        ex = e;
    }
}
if (!success) {
    throw new Exception(SERVER_ERROR, ex);
}
```

(a) Infer warns that object `ex` can be null and is dereferenced by `new Exception(...)` at line 13. This control-flow path is infeasible in practice if the property `ADDRESS_SERVER_RETRY_PROPERTY` is non-zero. If `success` is false, `ex` is guaranteed to be not `null` and thus marked uninteresting by a user.

```
String location = getLocation(LOGBACK_LOCATION);
try {
    ...
    LogbackConfigurator.configure(
        getResourceUrl(location));
    ...
} ...
```

(c) Infer warns that `location` can be null and is dereferenced by `ResourceUtils.getResourceUrl` at line 4. This null pointer dereference risk is low, as `getLocation` returns a value configured within the application. This warning is a low-priority warning.

```
String server = getProperty(SERVER, StringUtils.
    EMPTY);
if (!server.startsWith(HTTPS_PREFIX) &&
    !server.startsWith(HTTP_PREFIX)) {
    if (!InternetAddressUtil.containsPort(server
        )) {
            server = ...
    }
    server = HTTP_PREFIX + server;
}
```

(b) Infer warns that `server` can be null and is dereferenced at line 2. This warning uses code expression, `getProperty`, similar to Figure 3a, allowing a user to group similar warnings upon inspection of Figure 3a

```
String readFile(String path, String fileName) {
    ...
    File file = openFile(path);
    if (file.exists()) {
        ...
}
```

(d) A warning representing a genuine null pointer dereference risk. A call to `openFile` can return null, making object `file` null and `file.exists` to reference a null value. Since this warning is interesting, SWIRL infers rules with predicates such that they do not match previously marked uninteresting ones, e.g., a `String` return type.

Fig. 3: Figures 3a, 3b and 3c show three uninteresting warnings. These null pointer dereferences marked in yellow are not worthwhile to address, since these objects can be assumed to return valid values at runtime; With active feedback from a user on (a), and (b), SWIRL derives a summary rule to ignore uninteresting warnings in package `com.alibaba.nacos.client`. Then a user examines another warning (c) in the same package. When a user highlights `getProperty` in (b) to provide a hint, SWIRL refines the rule.

a) ***Example sensemaking with a containment relation.****:* Alice can view and provide feedback on multiple warnings simultaneously at the rule level. For instance, she notices that several warnings are clustered within the same package `com.alibaba.nacos.config.server`. Since these warnings all originate from the configuration server module, she realizes they are related to internal configuration handling and not to user-facing functionality. SWIRL proposes a rule based on this containment relation, grouping all warnings within the package together. After reviewing the matched warnings and confirming that none are of interest, Alice labels the entire group as *"Uninteresting"*, expediting her inspection by resolving dozens of warnings in one action.

b) ***Example sensemaking with salient code expressions.****:* For the remaining warnings, Alice labels a few more and SWIRL proposes a refined rule grouping warnings in the package `com.alibaba.nac- os.client`. As these warnings share similarities (see Figure 3c), she notices that the function call `getProperty()` is a marker of uninteresting warnings. She highlights `getProperty` in the code view to signal to SWIRL that this expression should be included when reformulating the rule. SWIRL then refines the rule to cover all warnings in `com.alibaba.nacos.client` that call `getProperty()`, enabling her to quickly summarize and dismiss other similar warnings.

III. Approach: Sensemaking with Interactive Review and Learning

This section introduces SWIRL's interactive rule-learning workflow (Fig. 4), casting summary-rule derivation as an active-learning task solved via inductive logic programming. SWIRL eases the process of sensemaking tool-generated warnings by proposing summary rules that group related warnings. Rules are iteratively refined.

A. Active learning-enabled rule formulation

We formulate the task of deriving summary rules as an **Active Learning** problem as follows:

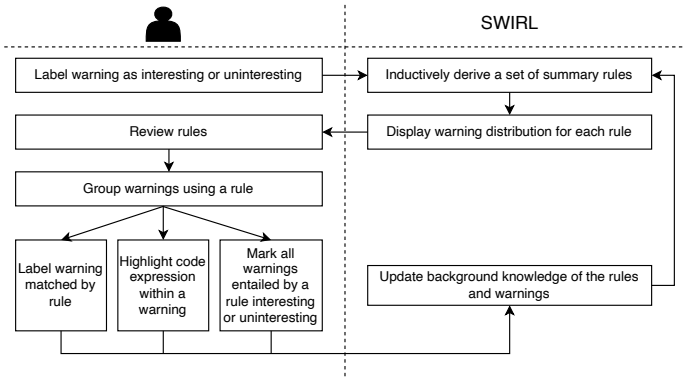


Fig. 4: SWIRL's workflow: A user can provide two modes of interactive feedback: (1) instance-level feedback for each warning, (2) marking all warnings as interesting or uninteresting per rule, and (3) clicking and highlighting code expressions in a code snippet view to guide rule inference. Based on feedback, SWIRL updates its candidate rules and displays a distribution of matched and unmatched warning instances for each summary rule.

Inputs:

1) Labels of each inspected warning. A warning, w, is labeled either as uninteresting ($w \in E^-$) or interesting ($w \in E^+$).
2) Background knowledge. A knowledge base with predicates on containment relationships and salient code expressions.

Output: SWIRL returns a set of summary rules, R. Each rule, $r \in R$, is a conjunction of predicates. The goal is for each rule to group related warnings. A warning matches a rule if every predicate in the rule is associated with it.

Active Learning. SWIRL has several requirements:

1) To minimize the time users spend analyzing individual warnings one by one, SWIRL should provide help early in the inspection process. Thus, even when the total number of warnings labeled "Uninteresting" ($|E^-|$) and "Interesting" ($|E^+|$), is much lower than the total number of warnings ($|W|$), SWIRL should be able to derive rules.
2) Rules derived from limited data are unlikely to accurately match the user's intent. Thus, their refinement is necessary to formulate meaningful rules. We favor starting with a general set of rules, which maximizes the number of matching, uninspected warnings.
3) Following Occam's Razor [16], SWIRL should propose as few rules as required.
4) No warnings are removed; rather custom rules are dynamically created by SWIRL to group and highlight the common characteristics of similar warnings.

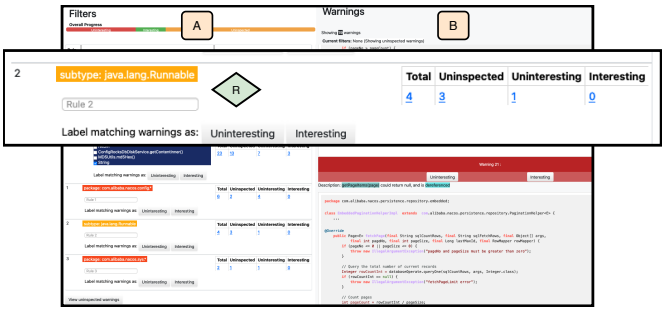
B. Inductive logic programming for deriving summary rules

As shown in Algorithm 1, SWIRL utilizes inductive logic programming [17] to derive a set of summary rules. SWIRL maintains a background knowledge base that is initialized by extracting facts of the implicated code's scope (package, class name), type hierarchy (subtyping/interfaces implemented) and other API signatures (return type, fields used) (line 1). The user provides feedback by marking each warning instance as interesting or uninteresting or by selecting a salient code expression for the warning. If the user provides instance-level feedback, the set of uninteresting warnings and interesting warnings are updated (lines 5–12). If the user highlights code expression as a hint in a code snippet window, the knowledge base is then updated by extracting new facts related to the code expression (lines 13–16). Note that we do not initialize the background knowledge with all possible code expressions, given the large space of possible rules, if every code expression is considered. Instead, SWIRL relies on the user to provide hints by highlighting salient code expressions.

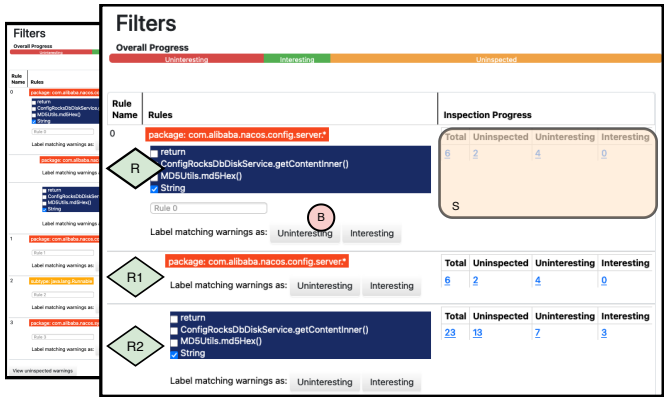
To derive a set of rules, SWIRL encodes the knowledge base as a logic program. The rules group potentially related warnings for user inspection. In our problem domain, a hypothesis is an assignment of predicates to rules such that the rule should not match the interesting examples (E^+), while matching as many uninteresting examples (E^-) (line 17). Ties between possible hypotheses are broken by the number of matched warnings, following the objective of maximizing the rules' generality. We use an off-the-shelf logic program solver [18] to search for candidate hypotheses. As shown in Figure 5a (R), SWIRL displays the rules, indicating the number of warnings matched by each rule.

Algorithm 1 Algorithm for deriving summary rules using Inductive Logic Programming [17], and updating the background knowledge based on the user's labeling and code highlighting feedback.

Input: $W \leftarrow$ set of warnings under inspection
1: $KB \leftarrow$ extract_containment_facts(W)
2: $E^+ \leftarrow \emptyset$
3: $E^- \leftarrow \emptyset$
4: **while** user still has inspections **do**
5: **if** user provided a new label **then**
6: $w, label \leftarrow$ FETCHNEWLABEL()
7: **if** $label$ is Uninteresting **then**
8: $E^- \leftarrow E^- \cup w$
9: **else if** $label$ is Interesting **then**
10: $E^+ \leftarrow E^+ \cup w$
11: **end if**
12: **end if**
13: **if** user selected code expression **then**
14: $code\ expression \leftarrow$ FETCHCODEEXPRESSION()
15: $KB \leftarrow$ UPDATEKB$(KB, W, code\ expression)$
16: **end if**
17: $R \leftarrow$ ILP_SOLVE(E^+, E^-, W, KB)
18: PRESENTTOUSER(R)
19: **end while**



(a) SWIRL has derived a candidate rule [see diamond R], which has a predicate checking for subtypes of `IntIterator`, which detects warnings reported in classes with `Runnable` as a supertype. In the individual warning view [see square B], each warning is shown with its matching rule numbers, alerting users that the given warning might be similar to a previously inspected warning.



(b) A rule [see diamond R] can combine both containment relations [see diamond R1] and code expressions [see diamond R2]. Beside each rule, the statistics [see rectangle S] are shown: the total number of warnings, the number of uninspected warnings, the number of warnings already marked as uninteresting vs. interesting. To expedite the sensemaking process, a user can provide feedback at the rule-level [see circle B].

Fig. 5: Containment and conjunction rules.

C. Rule design

Containment relationships. Prior work indicates that static-analysis alarms often cluster containment relationships (such as package, class, and type hierarchy) [12], [19]. SWIRL infers rules as a composite of predicates with the following structural relations.

1) Package: The package where the warning was reported (`package(X)`)
2) Classname: The name of the class where the warning was reported (`classname(X)`)
3) Return type: The return type of the method where the warning was reported (`rettype(X)`)
4) Fields used: The fields of the class used in the method (`fields(X)`)
5) Subtyping: The parent classes and interfaces of the class (`subtype(X)`)

Fry and Weimer report that static analysis reports in the same file and concerning the same program element (e.g. a particular function or variable) are likely duplicates that can be handled together [20]. Such evidence supports our decision to use containment predicates (package name, class name, parent type) as they capture a natural facet for grouping related warnings in the same context.

Salient code expressions. A user can select expressions to provide hints with code features associated with warnings, which can be specific method calls [21] or API usage conventions [22]. For a code expression highlighted by a user, SWIRL extracts the methods called (e.g., `getProperty()`), the types of the variables (e.g., `String`), the values of literals (e.g., `os.name`), control-flow elements, such as `if-then` statements, as well as conditionals such as a null-check. SWIRL encodes each code expression as a predicate, adding them to the background knowledge for rule inference. This grouping using salient expressions is motivated by prior findings highlighting the importance of capturing API usages through common method calls or similar expressions [23], [24].

Composite rules. SWIRL can derive rules that combine containment relationships with specific code expressions. For example, it can generate a rule to detect invocations of a given method "`getProperty()`" within the package `com.alibaba.nacos.core.cluster`, formally expressed as `package(com.alibaba.nacos.core.cluster)` and `codeElement(getProperty)`.

Conjoining containment predicates with salient code expressions yields more precise, developer-aligned groupings than any single facet; prior work shows that combining structural and code-level signals improves triage and reduces inspection effort (e.g., [20], [25]).

D. Distribution of matched inspected warnings

Sensemaking statistics. In SWIRL UI, the column "Inspection Progress" summarizes the distribution of warnings for each rule, as shown in Figure 5b [rectangle S] to indicate the sensemaking progress with formulated rules and to guide users in assessing if a candidate rule meaningfully groups related warnings. By clicking on the underlined numbers, a user can jump to the corresponding warnings matched by the rule.

Applying a rule to label all matching warnings. If a user believes that a rule accurately describes a group of similar uninteresting (or interesting) warnings, they can click the button beside "Label matching warnings as uninteresting (or interesting)" as shown in Figure 5b [see circle B]. A user can also rename a numbered rule with a more descriptive name to create meaningful abstraction for grouping related warnings.

IV. User Study

Our user study answers the following research questions:

1) RQ1. How much variation exists among the summary rules interactively formulated by different users?
2) RQ2. How does SWIRL influence developers' ability to identify commonalities between warnings, their cognitive load and their confidence in their inspections?
3) RQ3. What features of SWIRL are useful for formulating rule abstraction during the sensemaking process?

The goal of the user study is to assess if SWIRL enables participants to better articulate common characteristics of warnings that can inform downstream action (e.g., deprioritization, fixing the warnings, or policy updates). Accordingly, we measure the quality of groupings by assessing whether the participants could articulate common characteristics of related warnings, comparing SWIRL's effect on sensemaking against a baseline approach with inspecting warnings individually.

Baseline. To reflect how developers interact with static analysis tools in practice [3], [4], we constructed a downgraded version of SWIRL, where a user can browse warnings only one-by-one, similar to a typical interface of a warning list view bundled with Infer [15]. As shown in Figure 2, SpotBugs display warnings one by one for each analysis kind.

Participants. We ran a counter-balanced within-subject crossover study with 14 participants: 10 Ph.D. students, 2 master's students, and 2 industry developers, recruited from our CS department and professional networks. Experience ranged from 1–3 years (2), 4–6 years (3), 7–10 years (7), to $>$ 10 years (2). Exposing every participant to both tools in randomized order controls individual variability and enables paired comparisons with a modest sample size [26], [27]. Such within-subject studies with 8–16 participants are standard practice across Software Engineering [28], [29], [30], HCI [31], [32], [33], and sensemaking research [34], [35], [36], [37], [38].

Protocol. We conducted a 1-hour user study with each participant, involving both SWIRL and the baseline tool. The tool order and assigned tasks (Task A or Task B) were counterbalanced randomly. The study began with a 3-minute prestudy survey about the participants' background, followed by a 5-minute tutorial. Participants then completed 10 minutes of warm-up questions, inspecting a few null pointer dereference warnings from Infer. They proceeded to the main tasks, each lasting 10 minutes, where they inspected up to 25 warnings using the assigned tool. This time limit reflects typical usage of static analysis tools, where developers are reluctant to spend a significant amount of time analyzing the warnings, which we further verified in our pilot study. It was also not required for a user to review all warnings. After each task, participants filled out the NASA–TLX and post-task surveys (5 mins each). Finally, they completed a 12-minute post-study survey rating their experience with both tools and the usefulness of SWIRL's UI features.

Tasks. The user study consists of two tasks. For **Task A**, users inspected the warnings generated by Infer on Alibaba Nacos [8]. For **Task B**, users inspected warnings generated by SpotBugs on Apache Lucene-Solr [39]. These subject programs were used by prior work [40], [41], [42], [43], [44].

A. RQ1. Variation in user-formulated rules

Number and variance of derived rules. In total, 55 different rules were derived and used by the participants. Each participant formulated an average of 3.9 rules, and created at least one unique rule. This suggests that users develop *personalized interpretations* of warnings as they make sense of them, and that developers desire configurability. As such, a one-size-fits-all policy with universal and pre-defined groups may not work for summarizing warnings.

Rules	Participants
Warnings with code expression `CollectionUtils.isNotEmpty()`	P1, P4, P5, P11
Warnings with return type `boolean` matching `Iterator.*`	P2
Warnings in package `org.apache.solr.*` with return type `void`	P2
Warnings matching statement `File.is*`	P3, P6
Warnings with return type `Page<E>` for null object	P12
Warnings involving interface `NamedListInitializedPlugin`	P13

TABLE I: Example summary rules constructed by participants.

Table I shows examples of participant-formulated rules guided by SWIRL's active learning. A complete table is provided in our replication package [45]. The rules formulated by the participants capture a wide range of characteristics, including code expressions for third-party libraries (e.g., `CollectionUtils`), file handling (e.g., `File.listFiles()`), and specific API usage (e.g., use of `Iterators`). Other rules express containment relations in terms of the package name (e.g. `org.apache.solr.*`) or subtyping relations (e.g., `NamedListInitializedPlugin`). We observed utilization of salient code expressions for grouping, i.e., 4 out of 14 participants formulated a summary rule with code expression hint `CollectionUtils.isNotEmpty()`.

Types of rules. Participants used rules with different kinds, 51% of the derived rules were based on containment relationships (package name, class name, return type, used fields, and subtyping), and 40% were based on code expression hints highlighted by participants. The other 9% of the rules were composite rules consisting of both containment relationships and code expressions.

> **RQ1.** Of the 55 rules created, each participant contributed at least one unique rule, demonstrating individual approaches to sensemaking tool-generated warnings.

B. RQ2. Impact on sensemaking

Identifying the commonalities between uninteresting warnings. Participants were instructed to articulate possible root

causes/commonalities of the related warnings after each task. We found that participants could list 13% more common symptoms of related warnings (1.53 vs 1.35), when using SWIRL compared to the baseline on average. Participants also provided more detailed descriptions, writing an average of 4.6 sentences compared to just 3.1 sentences when using SWIRL vs. the baseline, indicating that they were able to better recognize similarities between related warnings. Participants using SWIRL pointed out specific code elements in their articulation more frequently than the baseline. For example, when using SWIRL, P6 pointed out that `isDirectory` acted as a guard for `listFiles`, making its null pointer dereference risk a low priority. P11 observed that some functions from third-party libraries can be assumed to handle null inputs gracefully (e.g., `Collections.isNotEmpty`).

Cognitive load. Results show that SWIRL was associated with significantly lower perceived *mental demand* compared to the baseline (-0.71 points on the questionnaire scale, $p = 0.043$). For *hurriedness*, participants also tended to report lower values with SWIRL (-0.65 points), though this difference did not reach significance ($p = 0.059$). No significant tool effects were observed for *effort*, *success*, or *frustration*. Descriptive means mirror these patterns in Figure 6: mental demand decreased (5.4→4.8), the perceived pace slowed (5.2→4.7), and frustration was lower (4.8→3.9). Table II further shows higher confidence with SWIRL (3.4 vs. 2.6) alongside a slight decrease in perceived ease of use (3.4 vs. 3.7). Taken together, these results suggest that SWIRL helped reduce participants' cognitive load when inspecting warnings, particularly in terms of mental demand.

RQ2. Participants using SWIRL better articulated commonalities among uninteresting warnings, listing 13% more symptoms and writing longer, more detailed descriptions. In doing so, they also experienced lower cognitive load: mental demand was ≈0.7 points lower than the baseline ($p = 0.043$), and confidence was higher (3.4 vs. 2.6, cf. Tab. II).

	SWIRL	Baseline
Confidence (1–5)	**3.43**	2.64
Ease of Use (1–5)	3.43	**3.69**

TABLE II: Using SWIRL, the participants reported higher confidence (3.4 vs. 2.6), with a slightly reduced ease of use (3.4 vs. 3.7).

C. RQ3. User perception of the interface features

Ease of use. Despite SWIRL's learning curve, participants found both the baseline and SWIRL easy to use (3.7 vs 3.5 on a 5-point Likert scale). P2 stated: "*I think if I got the hang of* SWIRL *and understood the filters a little bit better, going through these errors and marking them would be much easier ...*"

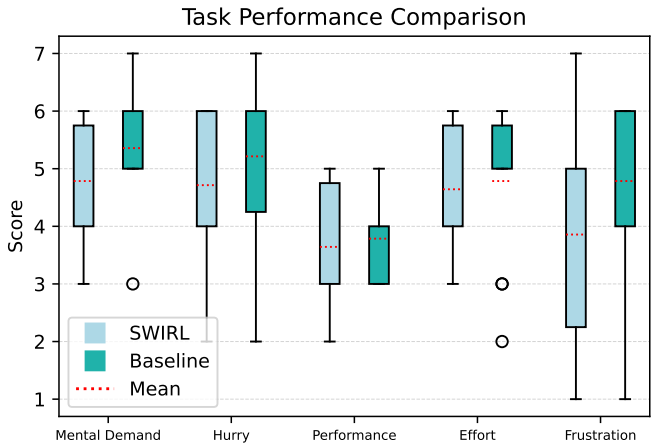


Fig. 6: Assessments of cognitive burden for SWIRL and the baseline by the participants. SWIRL reduced cognitive load for participants by decreasing the level of mental demand they experienced (5.4 to 4.8), slowing the perceived pace of the task (5.2 to 4.7), and lowering the frustration level (4.8 to 3.9).

Containment relationships. 10 out of 14 participants rated the feature of inferring rules with containment relations (package name, class name, used fields, return type, and subtyping) useful with a score of 3.2 out of 5. P3 wrote: "*Due to its similarity of those grouped alarms, I can understand the common root cause and skip the 'warm-up' phase in my decision-making.*"

Highlighting expressions as hints. 12 out of 14 participants perceived highlighting code expressions to be more useful (a score of 3.7 out of 5) as it produced groups of similar warnings. P4 wrote that this allowed them to "*easily check all similar warnings at once"* since "*they might have the same reasons behind.*"

Warning distribution. SWIRL displays the number of matched warnings for each rule, including the proportion of uninteresting warnings already labeled by the user. 11 out of 14 participants found this feature useful (a score of 3.8 out of 5), as it allowed them to determine "*which rules are more general and are worth looking at.*"

Label all warnings matched by a rule. Users can apply the same inspection decision to all warnings matched by the rule. 8 out of 14 participants found this feature useful (avg. 3.2/5), particularly when they were confident the rule captured only warnings sharing the same inspection decision. P9 said, "*Essentially the warnings are largely repetitive, as the static analyzer makes the same correct/mistaken decisions. So once the programmer has learned its pattern, it can be used to label all similar warnings.*"

In aggregation, 13 out of 14 participants found SWIRL useful. Moreover, they indicated that they would like to use SWIRL in the future (4.1 on a 5-point Likert scale).

RQ3. Participants found both containment relations and code expressions useful for constructing summary rules; they found being able to highlight concrete code expressions as hints useful for formulating custom abstraction.

V. Simulation

To complement and further verify the user study findings, we conducted a simulation study. While the user study demonstrated that SWIRL improves sensemaking quality and reduces cognitive load, it could not isolate the specific benefit of rule-level feedback over instance-level feedback at scale. The simulation addresses this by assessing the impact of inspection-order heuristics and feedback granularity (rule vs. instance) on the alignment of learned rules with user feedback, evaluated on the same data we used for **Task A**.

A. Simulated Behavior

In the user study, we observed that participants employ three heuristics to examine tool-generated warnings:

- *Heuristic 1.* Inspect **shorter** warnings first. Observed among participants [$P1$, $P2$, $P4$, $P5$, $P8$, $P9$, $P10$].
- *Heuristic 2.* Inspect warnings in terms of **similar API calls** first. Observed among participants [$P1$, $P2$, $P3$, $P5$, $P7$, $P8$, $P9$, $P10$].
- *Heuristic 3.* Inspect warnings located with **similar container names** (e.g., the same package name, the same class name, or the same file prefix). Observed among participants [$P2$, $P5$, $P9$].

In each interaction, the simulated user examines either one uninspected warning or an inferred rule. With a probability p, a user provides feedback at the rule level. For example, $p = 0.3$ means that a user provides rule-level feedback 30% of interactions, i.e., marking all warnings entailed by the rule as uninteresting. For providing feedback, we simulate a user making a decision for each of the selected warnings. All heuristics means using combined three heuristics in tandem. In other words, our simulated user combines Heuristics 1, 2, and 3 by randomly using one heuristic by selecting an uninspected warning for inspection.

B. Metric

Alignment. Alignment measures the proportion of rules aligned with the user's feedback and quantifies the benefit of rule-based summary. The higher it is, the better as it shows that rules reflect the user's views on labeling. If a rule r matches 5 warnings and 4 of them have the same label assignment as the user's labeling, we say r is 80% aligned with the user's labeling. We then measure the percentage of SWIRL's rules that are least k% aligned. We report our results with a target alignment threshold k of 80%, i.e., an aligned rule is at least 80% consistent with the user's feedback.

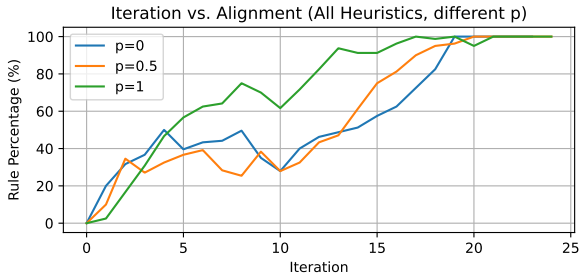


Fig. 7: Average alignment results over 20 simulation runs: effect of feedback probability p. When providing rule-level feedback ($p = 1$), the increase in alignment occurred at earlier iterations compared to providing only feedback to warnings ($p = 0$) or less frequent rule-level feedback ($p = 0.5$).

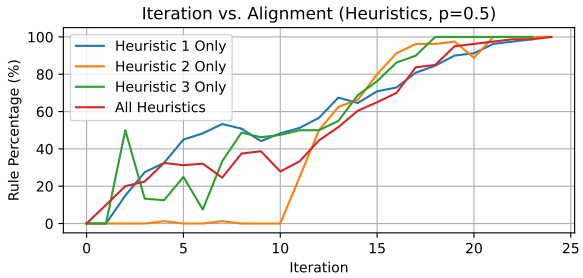


Fig. 8: Average alignment results over 20 simulation runs: effect of inspection-order heuristics.

C. Results

Figure 7 presents the alignment of the rules as the user iteratively interacts with SWIRL. Providing feedback at the warning level only ($p = 0$) frequently triggers rule refinement and produces noticeable fluctuations in alignment. In contrast, rule-level feedback ($p = 1$) yields a much smoother alignment. When providing rule-level feedback only ($p = 1$), the alignment score reaches 80% ($k = 80\%$) within 11.8^{th} interaction, the first time. When providing instance-level feedback only ($p = 0$), the alignment score reaches 80% the first time, within 17.8^{th} interaction.

Next, we investigate the effect of varying user behavior. We ran 20 simulations, while varying user behavior (Heuristic 1: shorter warnings first, Heuristic 2: similar API calls first, Heuristic 3: similar container names first, and 4: all three heuristics). Figure 8 shows that under different user behavior, rules become aligned with user feedback within 18 iterations. Inspecting warnings in the order of short length or based on the container names leads to more stable and smooth alignment improvement.

SWIRL's rules become aligned with user feedback as interaction progresses. Providing rule-level feedback results in faster alignment than providing warning-level feedback.

VI. Discussion

Inferring configurable and personalized rules. A high degree of variation exists among the rules formulated by participants with the help of SWIRL. This aligns with our design goal: SWIRL emphasizes grouping related warnings for sensemaking, not filtering them away. SWIRL achieves this by creating personalized rules based on each user's feedback. Accordingly, learned rules aim to maximize the concentration (and coverage) of similarly labeled instances within each group (e.g., many "Uninteresting"), yet do not require complete purity; mixed groups are acceptable and gradually refined as feedback accrues. In the post-study survey, participants indicated a preference for highlighting code expressions over pre-defined predicates (i.e., package and class names, subtyping, used fields, called methods, and return type), showing that participants appreciated the interactive features of SWIRL. Several participants also asked for even more customization, including defining rules from scratch. Future work can enhance more customization by enabling direct editing of rules with dual modalities of logic rules and natural-language descriptions.

In the current state of rules inferred by SWIRL, we observed positive feedback as noted by P8: *"These [Rules] help to avoid human bias when seeing the same repeated structure again. It eases the look up task for similar warnings from the same packages and classes thus narrowing down the search space.*" and by P9: *"This [Rule inference] is critical to the inductive reasoning in the tool."* However, some participants noted they preferred more granular rules, as noted by P6: *"The package containment rules didn't seem to help since the package rule was often too broad* `(e.g. org.*)`*."*

Sensemaking quality. A key finding of our user study is that SWIRL improved participants' ability to articulate commonalities among uninteresting warnings. Participants listed 13% more common symptoms (1.53 vs. 1.35) and wrote longer, more detailed descriptions (4.6 vs. 3.1 sentences on average) when using SWIRL compared to the baseline. This suggests that interactive summary rules do not merely organize warnings superficially, but enable developers to construct richer mental models of warning patterns. The ability to articulate commonalities is directly actionable: developers can use these insights to de-prioritize groups of similar warnings, update inspection policies, or communicate rationales to teammates.

Developer confidence. Participants reported greater confidence when using SWIRL (3.4 vs. 2.6 on a 5-point scale). This is notable given that static analysis tools are often perceived as generating too many false positives, leading to skepticism and tool abandonment [3]. By helping developers recognize patterns among uninteresting warnings, SWIRL may help rebuild trust in static analysis tools by making the inspection process feel more manageable and purposeful.

Efficiency of inspection. While SWIRL increased user confidence, some participants took longer to examine tool-generated warnings. On average, participants using the baseline inspected over 20% more warnings. Having to review the rules carefully slowed down the participants, which may indicate the care taken to reason about the commonalities among related warnings. P6 mentions "*efficiency gained* [using SWIRL] *was offset by the time I spent trying to get the inference engine to generate the rules I wanted*".

Utility of individual features. Among SWIRL's features, the warning distribution display was rated most useful (3.8/5, 11 out of 14 participants), followed by highlighting code expressions (3.7/5, 12 out of 14). These ratings suggest that both seeing how many warnings a rule covers and providing fine-grained code-level hints are central to the sensemaking workflow. The label-all feature was more situational, found useful by 8 out of 14 participants — most effective when participants were confident that a rule captured a coherent group of warnings.

Implications of the simulation. The simulation study shows that rule-level feedback leads to faster alignment than instance-level feedback (11.8 vs. 17.8 interactions to reach 80% alignment). This has a practical implication: in real workflows where warning sets can be large, the ability to confirm or reject an entire group at the rule level substantially reduces the number of interactions needed to align SWIRL's groupings with the developer's intent. This suggests that rule-level feedback is not merely a convenience feature, but a key driver of SWIRL's efficiency at scale.

Ease of use. There is an initial learning curve for using SWIRL compared to the examination of individual warnings. Nevertheless, tools perceived to be difficult to use can still provide value [46].

Failure Cases. SWIRL's rule inference algorithm is designed to group warnings based on the predicates present in self-contained functions and consistent user labeling. In the case of highly heterogeneous warning patterns with commonalities that span multiple procedures (e.g., taint flows or data races), or inconsistent user labeling, SWIRL may produce lower-quality rules.

Threats to validity. One threat to validity is related to the choice of bug types (i.e., analysis kind) used in our study. We focused on null pointer dereferences and resource leaks since these kinds of analysis are widely supported by commercial static analyzers. Grouping based on containment, type hierarchy, code similarity, and API methods and fields used in SWIRL should generalize to grouping other kinds of tool-generated warnings on code snippets. A second threat concerns the 10-minute cap per task. Short windows may bias participants toward shallow triage. We mitigated this by providing a tutorial and warm-up, using identical time limits for both conditions, and counterbalancing tool/order to control learning effects; still, longer or longitudinal use may reveal additional benefits. While our study included only 14 participants, studies that use a within-subject study with a crossover design require fewer participants [27], as the crossover minimizes variability [47].

VII. Related Work

Sensemaking. Prior sense-making systems (Selenite and Synergi for large text corpora [34], [35], Mesh for e-commerce trade-offs [48], Unakite for programming Q&A snippets [49], and decision-support tools such as Dreamsheets, Pail, and cost-structure models [37], [36], [50]) show that exposing similarities and contrasts accelerates insight, but are not applied to software engineering tasks. The closest work [38], [51], [52] highlights confidence and uncertainty cues only on a single code artifact at a time. SWIRL enables variation-driven sensemaking for triaging static-analysis warnings.

Filtering and reprioritizing. A large amount of work has focused on post-processing warnings generated from static bug finding tools [53], [54], [55], [56], [57], [41], [58], [44], which filter or re-prioritize generated warnings so that the uninteresting findings should appear last. Some of this work [41], [44], [40], [42] have shown that machine learning techniques could be effective at predicting which warnings are likely to be uninteresting. However, they fail to support developers for sensemaking the warnings. This can be helpful, but does not help developers establish accurate mental models to understand the alerts. The goal is not to bury uninteresting warnings, but to synthesize rules for developing insights and communicating rationales.

Clustering. Prior work on clustering static-analysis warnings identifies cause points (program locations whose imprecision triggers many alerts (e.g., a method spuriously deemed reachable) and employs interactive workflows to reduce triage effort [59], [60], [61], [62]. However, warnings are ignored for reasons other than analysis imprecision. For supporting sensemaking, SWIRL surfaces shared characteristics beyond analysis-imprecision program points: users can group related warnings by containment (e.g., package/class), by salient code expressions (e.g., specific API calls), and by their composition.

Manual inspection of tool-generated warnings. Other studies have proposed methods of inspecting warnings. Some analyzers focus on presenting proposed fixes [63] together with generated warnings. Mangal et al. present interactive static analysis, allowing users to make tradeoffs in the approximation made by the static analyzer [64]. Buckers et al. allow users to compare the warnings generated by different analyzers [19]. However, they do not support the sensemaking process necessary for developers to provide accurate feedback.

Ma et al. [65] categorize warnings by their bug types and the source and sink function calls. Unlike SWIRL, these categories (similar sources and sinks) are determined prior to users' inspections, and are not updated on-the-fly following the user's gradual and interactive feedback. Compared to SWIRL, the warnings are categorized without common code expressions at a fine granularity. Muske et al. [61] identify the cause points behind imprecise analyses. Their technique heuristically detects locations that are challenging for static analysis before posing queries to the user based on these locations. In contrast, SWIRL does not depend on pre-defined assumptions about what locations are challenging for static analysis, and can learn custom rules about the implicated code snippets with interactive user feedback.

Code pattern inference using human feedback. Studies have developed methods of inferring code patterns using human feedback [43], [66], [67], [68], [69], [70], Similar to these approaches, SWIRL includes a rule-learning component, but composes rules over two different modalities: (1) code organization in terms of containment, subtyping, subclassing, called methods, used fields, return type relationships and (2) code expressions harvested with interactive highlighting feedback for fine-granular rules.

Sensemaking of data. Many studies have proposed tools for sensemaking of large collections of data. Tempura [71] learns structural template for grouping textual data. SWIRL has similarities to PaTaT [72], a tool that learns patterns based on user-annotated codes as the user inspects data. These tools support human subjectivity and agency [73] in assigning labels to data, but both Tempura and PaTaT focus on textual data, while SWIRL is designed for inspecting warnings on code snippets reported by bug finding tools.

OverCode [74], Examplore [75], and ExampleNet [76] summarize a collection of code by providing a bird-eye view or a structure over variations in code. Instead of summarizing the variations in code, SWIRL highlights common patterns of tool-generated warnings by leveraging containment relationships and common code expressions, and enable refinements through interactive feedback.

VIII. Conclusion

In this paper, we propose SWIRL, which provides systematic support for personalized sensemaking of tool-generated warnings by harnessing user feedback with active learning. Our key insight is that automatically refined summary rules can enable users to contrast common characteristics among related warnings and that users' specific hints should be incorporated to re-infer summary rules.

We evaluate SWIRL with two different evaluation methods. First, our user study reveals that participants developed individualized, custom rules for grouping related warnings, confirming our insight that one-size-fits-all policy may not work during the cognitive, sensemaking process and that users desire configurability. When using SWIRL, users reported lower levels of frustration, higher levels of confidence, and could articulate more details about underlying similarities regarding uninteresting warnings. Second, our simulation study found that, by enabling users to provide feedback at the level of rules, SWIRL's resulting rules become aligned faster with labeling feedback.

IX. Data Availability

The replication package and study's data are available at https://github.com/UCLA-SEAL/SWIRL.

Acknowledgments

This work is supported by the National Science Foundation under grant numbers 2426162, 2106838, and 2106404. It is also supported in part by funding from Amazon and Samsung.

References

[1] N. Ayewah, W. Pugh, D. Hovemeyer, J. D. Morgenthaler, and J. Penix, "Using static analysis to find bugs," IEEE software, vol. 25, no. 5, pp. 22–29, 2008.

[2] C. Calcagno, D. Distefano, J. Dubreil, D. Gabi, P. Hooimeijer, M. Luca, P. O'Hearn, I. Papakonstantinou, J. Purbrick, and D. Rodriguez, "Moving fast with software verification," in NASA Formal Methods Symposium. Springer, 2015, pp. 3–11.

[3] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in 2013 35th International Conference on Software Engineering (ICSE). IEEE, 2013, pp. 672–681.

[4] M. Christakis and C. Bird, "What developers want and need from program analysis: an empirical study," in Proceedings of the 31st IEEE/ACM international conference on automated software engineering, 2016, pp. 332–343.

[5] H. Shen, J. Fang, and J. Zhao, "Efindbugs: Effective error ranking for findbugs," in Verification and Validation 2011 Fourth IEEE International Conference on Software Testing, Mar. 2011, p. 299–308. [Online]. Available: https://ieeexplore.ieee.org/document/5770619

[6] D. Tiganov, L. Nguyen Quang Do, and K. Ali, "Designing uis for static analysis tools: Evaluating tool design guidelines with swan," Queue, vol. 19, no. 4, pp. Pages 40:97–Pages 40:118, Sep. 2021. [Online]. Available: https://dl.acm.org/doi/10.1145/3487019.3487026

[7] F. Zampetti, S. Scalabrino, R. Oliveto, G. Canfora, and M. Di Penta, "How open source projects use static code analysis tools in continuous integration pipelines," in 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR). IEEE, 2017, pp. 334–344.

[8] Mar. 2024. [Online]. Available: https://github.com/alibaba/nacos

[9] J. Smith, L. N. Q. Do, and E. Murphy-Hill, "Why can't johnny fix vulnerabilities: A usability evaluation of static analysis tools for security," in Sixteenth Symposium on Usable Privacy and Security (SOUPS 2020). USENIX Association, Aug. 2020, pp. 221–238. [Online]. Available: https://www.usenix.org/conference/soups2020/presentation/smith

[10] Z. Guo, T. Tan, S. Liu, X. Liu, W. Lai, Y. Yang, Y. Li, L. Chen, W. Dong, and Y. Zhou, "Mitigating false positive static analysis warnings: Progress, challenges, and opportunities," IEEE Transactions on Software Engineering, vol. 49, no. 12, pp. 5154–5188, 2023.

[11] F. Marton, Necessary conditions of learning. Routledge, 2014.

[12] T. Kremenek, K. Ashcraft, J. Yang, and D. Engler, "Correlation exploitation in error ranking," ACM SIGSOFT Software Engineering Notes, vol. 29, no. 6, pp. 83–93, 2004.

[13] D. Cohn, L. Atlas, and R. Ladner, "Improving generalization with active learning," Machine Learning, vol. 15, no. 2, pp. 201–221, 1994. [Online]. Available: https://doi.org/10.1007/BF00993277

[14] B. Settles, "Active learning literature survey," 2009.

[15] "Infer explore," https://fbinfer.com/man/next/infer-explore.1.html/, 2023.

[16] A. Blumer, A. Ehrenfeucht, D. Haussler, and M. K. Warmuth, "Occam's razor," Information processing letters, vol. 24, no. 6, pp. 377–380, 1987.

[17] A. Cropper, S. Dumančić, R. Evans, and S. H. Muggleton, "Inductive logic programming at 30," Machine Learning, vol. 111, no. 1, pp. 147–172, 2022.

[18] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, and S. Thiele, "gringo, clasp, clingo, and iclingo," 2010.

[19] T. Buckers, C. Cao, M. Doesburg, B. Gong, S. Wang, M. Beller, and A. Zaidman, "Uav: Warnings from multiple automated static analysis tools at a glance," in 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, 2017, pp. 472–476.

[20] Z. P. Fry and W. Weimer, "Clustering static analysis defect reports to reduce maintenance costs," in 2013 20th Working Conference on Reverse Engineering (WCRE), 2013, pp. 282–291.

[21] R. van Tonder and C. L. Goues, "Tailoring programs for static analysis via program transformation," in Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, 2020, pp. 824–834.

[22] G. Uddin, B. Dagenais, and M. P. Robillard, "Analyzing temporal api usage patterns," in 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011). IEEE, 2011, pp. 456–459.

[23] T. Zhang, G. Upadhyaya, A. Reinhardt, H. Rajan, and M. Kim, "Are code examples on an online q&a forum reliable?: a study of API misuse on stack overflow," in 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE). IEEE, 2018, pp. 886–896.

[24] H. A. Nguyen, T. T. Nguyen, G. Wilson, A. T. Nguyen, M. Kim, and T. N. Nguyen, "A graph-based approach to api usage adaptation," SIGPLAN Not., vol. 45, no. 10, p. 302–321, Oct. 2010. [Online]. Available: https://doi.org/10.1145/1932682.1869486

[25] G. Liargkovas, E. Panourgia, and D. Spinellis, "Quieting the static: A study of static analysis alert suppressions," 2023. [Online]. Available: https://arxiv.org/abs/2311.07482

[26] S. Vegas, C. Apa, and N. Juristo, "Crossover designs in software engineering experiments: Benefits and perils," IEEE Transactions on Software Engineering, vol. 42, no. 2, p. 120–135, Feb. 2016.

[27] S. E. Chasins, E. L. Glassman, and J. Sunshine, "Pl and hci: better together," Communications of the ACM, vol. 64, no. 8, pp. 98–106, 2021.

[28] E. J. Arteaga Garcia, J. F. Nicolaci Pimentel, Z. Feng, M. Gerosa, I. Steinmacher, and A. Sarma, "How to support ml end-user programmers through a conversational agent," in Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, Feb. 2024, p. 1–12. [Online]. Available: https://dl.acm.org/doi/10.1145/3597503.3608130

[29] H. J. Kang, K. Wang, and M. Kim, "Scaling code pattern inference with interactive what-if analysis," in Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, Apr. 2024, p. 1–12. [Online]. Available: https://dl.acm.org/doi/10.1145/3597503.3639193

[30] M. Ganji, S. Alimadadi, and F. Tip, "Code coverage criteria for asynchronous programs," in Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ser. ESEC/FSE 2023. New York, NY, USA: Association for Computing Machinery, Nov. 2023, p. 1307–1319. [Online]. Available: https://doi.org/10.1145/3611643.3616292

[31] A. Horvath, B. Myers, A. Macvean, and I. Rahman, "Using annotations for sensemaking about code," in UIST'22. Bend OR USA: ACM, Oct. 2022, p. 1–16. [Online]. Available: https://dl.acm.org/doi/10.1145/3526113.3545667

[32] S. Suh, B. Min, S. Palani, and H. Xia, "Sensecape: Enabling multilevel exploration and sensemaking with large language models," in Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology, ser. UIST '23. New York, NY, USA: Association for Computing Machinery, Oct. 2023, p. 1–18. [Online]. Available: https://dl.acm.org/doi/10.1145/3586183.3606756

[33] M. Huh and A. Pavel, "Designchecker: Visual design support for blind and low vision web developers," in Proceedings of the 37th Annual ACM Symposium on User Interface Software and Technology, ser. UIST '24. New York, NY, USA: Association for Computing Machinery, Oct. 2024, p. 1–19. [Online]. Available: https://dl.acm.org/doi/10.1145/3654777.3676369

[34] M. X. Liu, T. Wu, T. Chen, F. M. Li, A. Kittur, and B. A. Myers, "Selenite: Scaffolding online sensemaking with comprehensive overviews elicited from large language models," in Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems, ser. CHI '24. New York, NY, USA: Association for Computing Machinery, May 2024, p. 1–26. [Online]. Available: https://dl.acm.org/doi/10.1145/3613904.3642149

[35] H. B. Kang, T. Wu, J. C. Chang, and A. Kittur, "Synergi: A mixed-initiative system for scholarly synthesis and sensemaking," in Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology, ser. UIST '23. New York, NY, USA: Association for Computing Machinery, Oct. 2023, p. 1–19. [Online]. Available: https://dl.acm.org/doi/10.1145/3586183.3606759

[36] J. D. Zamfirescu-Pereira, E. Jun, M. Terry, Q. Yang, and B. Hartmann, "Beyond code generation: Llm-supported exploration of the program design space," Mar. 2025, arXiv:2503.06911 [cs]. [Online]. Available: http://arxiv.org/abs/2503.06911

[37] S. G. Almeda, J. Zamfirescu-Pereira, K. W. Kim, P. Mani Rathnam, and B. Hartmann, "Prompting for discovery: Flexible sense-making for ai art-making with dreamsheets," in Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems, ser. CHI '24. New York, NY, USA: Association for Computing Machinery, May 2024, p. 1–17. [Online]. Available: https://dl.acm.org/doi/10.1145/3613904.3642858

[38] J. D. Weisz, M. Muller, S. Houde, J. Richards, S. I. Ross, F. Martinez, M. Agarwal, and K. Talamadupula, "Perfection not required? human-ai partnerships in code translation," in 26th International Conference on Intelligent User Interfaces, Apr. 2021, p. 402–412, arXiv:2104.03820 [cs]. [Online]. Available: http://arxiv.org/abs/2104.03820

[39] Mar. 2024. [Online]. Available: https://github.com/apache/lucene-solr

[40] A. Kharkar, R. Z. Moghaddam, M. Jin, X. Liu, X. Shi, C. Clement, and N. Sundaresan, "Learning to reduce false positives in analytic bug detectors," in Proceedings of the 44th International Conference on Software Engineering, 2022, pp. 1307–1316.

[41] J. Wang, S. Wang, and Q. Wang, "Is there a" golden" feature set for static warning identification? an experimental evaluation," in Proceedings of the 12th ACM/IEEE international symposium on empirical software engineering and measurement, 2018, pp. 1–10.

[42] X. Yang, Z. Yu, J. Wang, and T. Menzies, "Understanding static code warnings: An incremental ai approach," Expert Systems with Applications, vol. 167, p. 114134, 2021.

[43] H. J. Kang and D. Lo, "Active learning of discriminative subgraph patterns for api misuse detection," IEEE Transactions on Software Engineering, vol. 48, no. 8, pp. 2761–2783, 2021.

[44] R. Yedida, H. J. Kang, H. Tu, X. Yang, D. Lo, and T. Menzies, "How to find actionable static analysis warnings: A case study with findbugs," IEEE Transactions on Software Engineering, 2023.

[45] [Online]. Available: https://github.com/UCLA-SEAL/SWIRL/blob/master/code/data/user_study/All%20Rules.pdf

[46] A. Sarkar, "Should computers be easy to use? questioning the doctrine of simplicity in user interface design," in Extended Abstracts of the 2023 CHI Conference on Human Factors in Computing Systems, 2023, pp. 1–10.

[47] S. Vegas, C. Apa, and N. Juristo, "Crossover designs in software engineering experiments: Benefits and perils," IEEE Transactions on Software Engineering, vol. 42, no. 2, pp. 120–135, 2015.

[48] J. C. Chang, N. Hahn, and A. Kittur, "Mesh: Scaffolding comparison tables for online decision making," in Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology, ser. UIST '20. New York, NY, USA: Association for Computing Machinery, Oct. 2020, p. 391–405. [Online]. Available: https://dl.acm.org/doi/10.1145/3379337.3415865

[49] M. X. Liu, J. Hsieh, N. Hahn, A. Zhou, E. Deng, S. Burley, C. Taylor, A. Kittur, and B. A. Myers, "Unakite: Scaffolding developers' decision-making using the web," in Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology, ser. UIST '19. New York, NY, USA: Association for Computing Machinery, Oct. 2019, p. 67–80. [Online]. Available: https://dl.acm.org/doi/10.1145/3332165.3347908

[50] D. M. Russell, M. J. Stefik, P. Pirolli, and S. K. Card, "The cost structure of sensemaking," in Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems, ser. CHI '93. New York, NY, USA: Association for Computing Machinery, May 1993, p. 269–276. [Online]. Available: https://dl.acm.org/doi/10.1145/169059.169209

[51] H. Vasconcelos, G. Bansal, A. Fourney, Q. V. Liao, and J. W. Vaughan, "Generation probabilities are not enough: Uncertainty highlighting in ai code completions," ACM Transactions on Computer-Human Interaction, p. 3702320, Oct. 2024, arXiv:2302.07248 [cs].

[52] C. Spiess, D. Gros, K. S. Pai, M. Pradel, M. R. I. Rabin, A. Alipour, S. Jha, P. Devanbu, and T. Ahmed, "Calibration and correctness of language models for code," no. arXiv:2402.02047, Aug. 2024, arXiv:2402.02047 [cs]. [Online]. Available: http://arxiv.org/abs/2402.02047

[53] S. Kim and M. D. Ernst, "Prioritizing warning categories by analyzing software history," in Fourth International Workshop on Mining Software Repositories (MSR'07: ICSE Workshops 2007). IEEE, 2007, pp. 27–27.

[54] Q. Hanam, L. Tan, R. Holmes, and P. Lam, "Finding patterns in static analysis alerts: improving actionable alert ranking," in Proceedings of the 11th working conference on mining software repositories, 2014, pp. 152–161.

[55] S. Heckman and L. Williams, "A model building process for identifying actionable static analysis alerts," in 2009 International conference on software testing verification and validation. IEEE, 2009, pp. 161–170.

[56] J. R. Ruthruff, J. Penix, J. D. Morgenthaler, S. Elbaum, and G. Rothermel, "Predicting accurate and actionable static analysis warnings: an experimental approach," in Proceedings of the 30th international conference on Software engineering, 2008, pp. 341–350.

[57] H. Shen, J. Fang, and J. Zhao, "Efindbugs: Effective error ranking for findbugs," in 2011 Fourth IEEE International conference on software testing, verification and validation. IEEE, 2011, pp. 299–308.

[58] H. J. Kang, K. L. Aw, and D. Lo, "Detecting false alarms from automatic static analysis tools: How far are we?" in Proceedings of the 44th International Conference on Software Engineering, 2022, pp. 698–709.

[59] T. Muske and A. Serebrenik, "Survey of approaches for postprocessing of static analysis alarms," ACM Comput. Surv., vol. 55, no. 3, Feb. 2022. [Online]. Available: https://doi.org/10.1145/3494521

[60] I. Dillig, T. Dillig, and A. Aiken, "Automated error diagnosis using abductive inference," ACM SIGPLAN Notices, vol. 47, no. 6, pp. 181–192, 2012.

[61] T. Muske and U. P. Khedker, "Cause points analysis for effective handling of alarms," in 2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE). IEEE, 2016, pp. 173–184.

[62] X. Zhang, R. Grigore, X. Si, and M. Naik, "Effective interactive resolution of static analysis alarms," Proceedings of the ACM on Programming Languages, vol. 1, no. OOPSLA, pp. 1–30, 2017.

[63] T. Barik, Y. Song, B. Johnson, and E. Murphy-Hill, "From quick fixes to slow fixes: Reimagining static analysis resolutions to enable design space exploration," in 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, 2016, pp. 211–221.

[64] R. Mangal, X. Zhang, A. V. Nori, and M. Naik, "A user-guided approach to program analysis," in Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, 2015, pp. 462–473.

[65] X. Ma, J. Yan, J. Yan, and J. Zhang, "Reorganizing and optimizing post-inspection on suspicious bug reports in path-sensitive analysis," in 2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS). IEEE, 2019, pp. 260–271.

[66] T. Zhang, M. Song, J. Pinedo, and M. Kim, "Interactive code review for systematic changes," in 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, vol. 1. IEEE, 2015, pp. 111–122.

[67] A. Sivaraman, T. Zhang, G. Van den Broeck, and M. Kim, "Active inductive logic programming for code search," in 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). IEEE, 2019, pp. 292–303.

[68] C. Wang, P. Yao, W. Tang, G. Fan, and C. Zhang, "Synthesizing conjunctive queries for code search," in 37th European Conference on Object-Oriented Programming, 2023.

[69] H. J. Kang, K. Wang, and M. Kim, "Scaling code pattern inference with interactive what-if analysis," in IEEE/ACM International Conference on Software Engineering (ICSE), 2014.

[70] P. Garg and S. H. Sengamedu, "Synthesizing code quality rules from examples," Proceedings of the ACM on Programming Languages, vol. 6, no. OOPSLA2, pp. 1757–1787, 2022.

[71] T. Wu, K. Wongsuphasawat, D. Ren, K. Patel, and C. DuBois, "Tempura: Query analysis with structural templates," in Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems, 2020, pp. 1–12.

[72] S. A. Gebreegziabher, Z. Zhang, X. Tang, Y. Meng, E. L. Glassman, and T. J.-J. Li, "Patat: Human-ai collaborative qualitative coding with explainable interactive rule synthesis," in Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems, 2023, pp. 1–19.

[73] J. A. Jiang, K. Wade, C. Fiesler, and J. R. Brubaker, "Supporting serendipity: Opportunities and challenges for human-ai collaboration in qualitative analysis," Proceedings of the ACM on Human-Computer Interaction, vol. 5, no. CSCW1, pp. 1–23, 2021.

[74] E. L. Glassman, J. Scott, R. Singh, P. J. Guo, and R. C. Miller, "Overcode: Visualizing variation in student solutions to programming problems at scale," ACM Transactions on Computer-Human Interaction (TOCHI), vol. 22, no. 2, pp. 1–35, 2015.

[75] E. L. Glassman, T. Zhang, B. Hartmann, and M. Kim, "Visualizing api usage examples at scale," in Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems, 2018, pp. 1–12.

[76] L. Yan, E. L. Glassman, and T. Zhang, "Visualizing examples of deep neural networks at scale," in Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems, 2021, pp. 1–14.