Toward Declarative Auditing of Java Software for Graceful Exception Handling

Leo St. Amour
Virginia Tech
Blacksburg, USA
lstamour@vt.edu

Eli Tilevich
Virginia Tech
Blacksburg, USA
tilevich@cs.vt.edu

Abstract

Despite their language-integrated design, Java exceptions can be difficult to use effectively. Although Java exceptions are syntactically straightforward, negligent practices often result in code logic that is not only inelegant but also unsafe. This paper explores the challenge of auditing Java software to enhance the effectiveness and safety of its exception logic. We revisit common anti-patterns associated with Java exception usage and argue that, for auditing, their detection requires a more nuanced approach than mere identification. Specifically, we investigate whether reporting such anti-patterns can be prioritized for subsequent examination. We prototype our approach as Händel, in which anti-patterns and their priority, or weight, are expressed declaratively using probabilistic logic programming. Evaluation with representative open-source code bases suggests Händel's promise in detecting, reporting, and ranking the anti-patterns, thus helping streamline Java software auditing to ensure the safety and quality of exception-handling logic.

CCS Concepts: • **Software and its engineering** → Automated static analysis; **Software safety**; **Error handling and recovery**; • **Computing methodologies** → **Probabilistic reasoning**; *Logic programming and answer set programming*.

Keywords: Software Auditing, Static Program Analysis, Probabilistic Reasoning, Exceptions, Java, Logic Programming

ACM Reference Format:
Leo St. Amour and Eli Tilevich. 2024. Toward Declarative Auditing of Java Software for Graceful Exception Handling. In *Proceedings of the 21st ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR '24), September 19, 2024, Vienna, Austria.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3679007.3685057

MPLR '24, September 19, 2024, Vienna, Austria

ACM ISBN 979-8-4007-1118-3/24/09
https://doi.org/10.1145/3679007.3685057

1 Introduction

Java was not the first mainstream programming language with a built-in exception-handling mechanism. Dating back to the 1960s [35], several prior languages, including Ada [28] and C++ [26], supported handling runtime errors and exceptional conditions in a structured manner. Drawing inspiration from these languages, the design of Java has reconsidered several aspects of exceptions, including a standardized exception hierarchy, checked exceptions, and strict exception type checking [24]. After nearly 30 years, Java has experienced enormous success and now rules the world of enterprise software with billions of lines of legacy code. Unfortunately, analyzing legacy Java code reveals that exceptions have become a double-edged sword. Although it promotes the intended structured handling of exceptional conditions, it also allows undisciplined and ill-conceived programming practices [5, 6, 12, 17]. Unless following a strict set of coding principles, when it comes to exceptions, Java programmers are prone to exhibit common poor coding practices—often referred to as *anti-patterns* [9, 19].

Before a code base can be included in high-stakes environments, such as government or critical infrastructure systems, it must be audited for adherence to safety standards. Software auditing verifies and validates that a code base complies with the necessary standards and meets all its baseline requirements [10]. Unfortunately, auditing large code bases is notoriously time-consuming and tedious, requiring significant resources and expertise [32]. Consequently, auditing can greatly benefit from automated tools and processes [16].

Exception handling needs auditing as it fulfills the critical role of addressing exceptional runtime conditions. Certain features of Java exceptions make code susceptible to anti-patterns. Anders Hejlsberg, the lead C# architect, points out in an interview that "checked exceptions become such an irritation that people completely circumvent the feature… checked exceptions have actually degraded the quality of the system in the large" [48]. Indeed, Java programmers often circumvent the necessity to handle checked exceptions by creating empty catch clauses or catching a generic superclass exception type. Extensive research has codified such Java exception anti-patterns and studied their prevalence in legacy code bases [5, 6, 12, 38, 44, 53].

This paper focuses on the challenge of auditing Java software to ensure the effectiveness and safety of exception

```
public void myCreateBucket(String name) {
  S3Client s3 = newS3Client();
  try {
    CreateBucketRequest req = newCBRequest(name);
    s3.createBucket(req);
    // throws BucketAlreadyExistsException,
    //        BucketAlreadyOwnedByYouException,
    //        AwsServiceException,
    //        SdkClientException, and S3Exception
      ...
  } catch (SdkException e) {
    ... // handle exception appropriately
  }
  s3.close();
}
```

Figure 1. Code snippet for facilitating the exposition and definitions of Java exception anti-patterns; adapted from [2].

handling. Auditing tools that report many false positives cause information overload, particularly for large commercial code bases. With unlimited resources, an auditor could examine each reported case to confirm its validity. However, this practice would be infeasible in realistic settings. When it comes to auditing, detecting exception anti-patterns can benefit from a more nuanced treatment than the current methods that report their findings using boolean logic.

To address the challenges of boolean reporting, we introduce a novel approach that leverages probabilistic reasoning to detect and weigh possible anti-patterns. The weighted results can be used to guide further manual inspection. We prototype our approach as Händel, which concisely expresses anti-patterns in ProbLog, a probabilistic dialect of Prolog. This design enables declarative specifications that can be easily examined and tweaked. Despite the interpretive nature of ProbLog execution, Händel shows promising performance and memory consumption characteristics. An evaluation with representative Java benchmark applications demonstrates the potential of probabilistic reasoning for reporting suspected exception anti-patterns as well as Händel's ability to point out the likelihood of their presence. This paper makes the following contributions:

- We introduce a novel approach for reporting exception anti-patterns based on probabilistic reasoning to guide subsequent auditing efforts.
- We describe our prototype implementation, Händel, an extensible analysis framework that leverages probabilistic logic programming to identify potential exception anti-patterns; in Händel, analysis results are weighed via highly configurable ProbLog predicates.
- We assess our approach's suitability for auditing by applying Händel to a set of representative Java benchmark applications.

2 Exception-handling Anti-patterns

Previous works have established a Java exception-handling anti-pattern taxonomy [5]. This section revisits common anti-patterns defined in this taxonomy. Consider the Java method `myCreateBucket` depicted in Figure 1. This code snippet uses the Amazon AWS Java SDK [3]. Specifically, it creates a new bucket by using the API to interact with the Amazon S3 service. The API `createBucket` method throws numerous exceptions, as stated in the listing as comments. All these exceptions are sub-classes of the base class `SdkException`. This inheritance relationship makes it possible to use the base class exception in the `catch` clause for handling all possible exceptions.

Whether this code snippet matches an anti-pattern cannot be determined definitively. In some contexts, using the base class is appropriate for the desired level of exception handling. In others, it may be necessary to handle each of the potentially thrown exception sub-classes specially. In particular, from an auditing standpoint, we might need to be able to specify the degree to which a code base matches an anti-pattern. The reported degrees would then prioritize subsequent manual auditing. We next revisit common exception anti-patterns and argue that boolean reporting logic might be unnecessarily rigid for software auditing.

2.1 Anti-pattern 1: Catch Generic / Over-Catch

One of the most common Java exception anti-patterns is *Catch Generic* or *Over-Catch* (AP1), in which a handler catches an exception type that is a supertype of the thrown exceptions [38]. This type relationship is called subsumption [1]. This anti-pattern branches into two distinct types: (1) "catch generic", a program catching a high-level, generic exception type (i.e., Throwable, Exception, Error, or RuntimeError); (2) subsumption, or "over-catch" [53], a handler catching multiple different lower-level exceptions [5]. In both cases, the anti-pattern reflects that the caught exception type is a super-class of the types of exceptions thrown. As a result, the exception handler is less likely to appropriately account for the nuances of each possible sub-class of exception. For example, in our motivating example, we might want to be able to distinguish between `BucketAlreadyExistsException` and `BuckedAlreadyOwnedByYouException`, as we want to ensure that these particular exceptional conditions are handled specially. At the same time, it may be acceptable for the remaining exceptions to be handled generically.

2.2 Anti-pattern 2: Throws Generic / Over-Throws

Another Java exception anti-pattern is *Throws Generic* (AP2), in which a method's "throws" statement propagates a generic exception type [5]. Similar to AP1, this anti-pattern derives from using a super-class exception type. Because the specific thrown exception types are lost, any handlers that catch the thrown exceptions become incapable of specializing

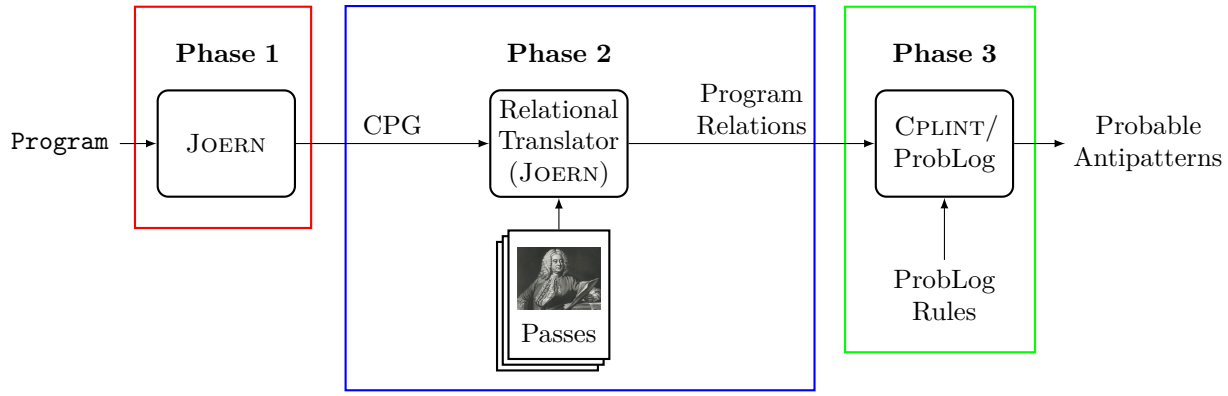

Figure 2. HÄNDEL system overview and data-flow diagram

their handling. Inspired by the over-catch anti-pattern, we propose a more encompassing *Over-Throws* anti-pattern in which the type specified in a throws statement subsumes the types of exceptions thrown. For example, the control flow of `createBucket` could pass through a method that propagates `SdkException`, which may or may not be problematic depending on the context.

2.3 Anti-pattern 3: Unhandled Exceptions

Yet another anti-pattern we consider is *Unhandled Exception* (AP3), in which a handler fails to catch all reachable exceptions [5, 44]. This anti-pattern describes the practice of implementing an exception handler that reaches but does not catch a thrown unchecked exception. By definition, the Java semantics do not require unchecked exceptions to be handled. However, within certain contexts, it may be necessary to handle such runtime exceptions gracefully. For example, assume the `createBucket` method in our motivating example additionally threw an `IllegalArgumentException`. If this exception is not handled, the program will terminate prematurely. Software auditors might want to ensure that specific unchecked exceptions are caught so that the software does not fail within a high-stakes environment.

3 HÄNDEL's Design and Implementation

Auditing contexts may differ greatly, depending on the software domain, deployment environment, and security/privacy restrictions. In light of these observations, our design must exhibit high degrees of transparency and configurability. The key insight of our design lies in employing probabilistic logic programming to specify our analyses. Transparency is achieved through comprehensible logical rules that specify anti-pattern detection. Configurability is achieved through special or custom logic predicates.

We prototype our approach as HÄNDEL, which draws inspiration from the renowned baroque composer George Frideric Händel. Just as Händel's compositions are celebrated for their elegance, precision, and gracefulness, HÄNDEL promotes these qualities in Java exception-handling code. Figure 2 presents an overview of the HÄNDEL framework, comprising three distinct phases, whose implementations we detail next.

3.1 Phase I: Generating Code Property Graph

In HÄNDEL's first phase, a program is converted into a code property graph (CPG). A CPG is a powerful program representation that combines a program's abstract syntax tree (AST), control flow graph (CFG), and program dependency graph (PDG) into a single, joint structure [52]. The AST represents the program's source code and syntactic structure; the CFG captures how execution flows between program statements; and the PDG encodes data dependencies throughout the program. A CPG provides the advantages of all three graphs in a single, convenient representation. While CPGs were originally designed for describing and identifying software vulnerabilities, its applicability extends beyond security-based analyses.

Because they provide a rich expression of a program's properties, we adopted CPGs as an intermediate representation for our analyses. The AST sub-graph contains Java exception-handling constructs. Each try/catch/finally structure has a corresponding sub-tree in the AST. A "try" control structure is at the sub-tree's root, and each block comprises the child nodes. We build our anti-pattern analyses by combining the exception-handling structures expressed in the AST with the control-flow relationships encoded in the CFG sub-graph. To obtain a CPG representation of a program, HÄNDEL uses JOERN, a CPG framework [22].

3.2 Phase II: Translating CPGs into ProbLog Facts

HÄNDEL's second phase translates a given CPG into ProbLog facts representing the program's control-flow and exception-handling relationships. As depicted in Figure 2, this phase accepts a set of HÄNDEL passes as input. Each pass translates a targeted set of program constructs into ProbLog facts.

3.2.1 Control-flow Pass. This pass outputs a set of relations describing how execution transitions between program statements and methods. Specifically, it translates a program's CFG and constructs its probabilistic CFG (Prob-CFG), a weighted CFG, in which an edge's weight represents the probability of following that edge [41]. The logic for applying probabilities to control-flow edges is based on branch selectivity [40]. Each branch condition in the program is converted into satisfiability modulo theory (SMT) constraints and solved using the automata-based model counter (ABC) SMT solver [4]. This pass outputs a set of facts representing an *intra-procedural* CFG/Prob-CFG for each method. The declarative ProbLog rules then infer the set of *inter-procedural* edges, thus demonstrating an additional advantage of structuring this analysis using probabilistic logic programming. Specifically, this pass outputs the following facts:

- `method(Entry,Name)`: CFG node `Entry` is the entry point for method `Name`.
- `cfg_edge(X,Y)`: Edge between CFG nodes `X` and `Y`.
- `P::prob_cfg_edge(X,Y)`: Edge between CFG nodes `X` and `Y` with probability `P`.
- `calls(Meth,Callee,Site)`: `Meth` calls `Callee` at CFG node `Site`.
- `returns(Meth,Site)`: `Meth` returns at CFG node `Site`.

3.2.2 Exception-handling Pass. This pass traverses the CPG and identifies nodes relevant to exception-handling constructs. Specifically, it identifies exception-handling control structures (i.e., try and catch blocks), caught exception types, thrown exceptions, and propagated exceptions. This pass outputs the following exception-handling facts:

- `method_throws(Meth,Exc)`: `Meth` propagates `Exc`.
- `throws(X,Exc)`: `Exc` is thrown at CFG node `X`.
- `catches(Try,Catch,Exc)`: Control structure `Try` handles `Exc` in block `Catch`.
- `in_try(X,Try)`: CFG node `X` is in the try block of control structure `Try`.
- `in_catch(X,Try)`: CFG node `X` is in the catch block of control structure `Try`.
- `subclass(T1,T2)`: Type `T1` is a sub-class of `T2`.

3.3 Phase III: Identifying Probable Anti-patterns

Händel's third and final phase employs ProbLog to identify potential anti-patterns and their corresponding weights. As its logic engine, Händel utilizes Cplint [36, 37], an implementation of ProbLog provided as a library for SWI-Prolog[51]. We selected Cplint due to its full support of the ProbLog syntax and the robustness of SWI-Prolog.

Händel infers inter-procedural control edges and paths from the control flow facts extracted in Phase II. The rules in Figure 3 specify the relationships that infer inter-procedural control flow edges and paths between nodes `X` and `Y`.

The `icfg_edge` rule on line one represents intra-procedural edges. Any existing CFG edges not originating from a call should be included in the inter-procedural CFG. The rule on line two represents edges between functions. There is an edge between nodes `X` and `Y` if `X` is a call site and `Y` is the callee. The rule on line three establishes a back-edge from a called function to its call site. There is an edge between nodes `X` and `Y` if `X` is the return site of a function that was called by the node immediately preceding `Y`. The `icfg_path` rules on lines six and seven demonstrate how to infer whether a path between `X` and `Y` exists using the `icfg_edge` relationships. We introduce a nearly identical set of rules for `prob_icfg_edge` and `prob_icfg_path` by replacing instances of `cfg_edge` with `prob_cfg_edge`. These rules are integrated into additional rules that define our anti-patterns.

```
1  icfg_edge(X,Y) :- cfg_edge(X,Y), \+calls(_,_,X).
2  icfg_edge(X,Y) :- calls(_,Y,X).
3  icfg_edge(X,Y) :- calls(_,M,Z), returns(M,X),
4      cfg_edge(Z,Y).
5
6  icfg_path(X,Y) :- icfg_edge(X,Y).
7  icfg_path(X,Y) :- icfg_edge(X,Z), icfg_path(Z,Y).
```

Figure 3. Inter-procedural edge and path inference rules

```
1  P::exception_distance(N) :- P is 1-(1/N)+0.2.
2
3  antipattern1(Catch,CaughtExc,Throw,ThrownExc,N) :-
4      throws(Throw,ThrownExc),
5      catches(Try,Catch,CaughtExc),
6      is_subclass(ThrownExc,CaughtExc,N),
7      exception_distance(N),
8      in_try(TryNode,Try),
9      in_catch(CatchNode,Try),
10     icfg_path(TryNode,Throw),
11     icfg_path(Throw,CatchNode).
```

Figure 4. Specification for AP1: catch generic / over-catch

Figure 4 presents the ProbLog rule for AP1, which identifies whether an exception handler `Catch` is over-catching. If an exception is thrown along a path between a try block and its catch block, and the thrown exception is a sub-class of the caught exception, then the handler, `Catch`, may match AP1. The result is weighted via the `exception_distance` predicate. For our purposes, the further apart the two exceptions are in the type hierarchy, the more likely it is that AP1 has been matched. Depending on the audit, the weight predicate can be modified. For example, `exception_distance` can be updated to use a different formula or replaced with a different predicate that represents the targeted standard.

For brevity, we omit the AP2 and AP3 specifications. However, they are equally as comprehensible and configurable as AP1. AP2 is also defined by type subsumption, so it uses `exception_distance` as its weight predicate. The weight predicate for AP3 is derived from the Prob-CFG as the cumulative probability of the edges along a `prob_icfg_path`. If a program throws an exception along a typical—or probable—path, the exception's graceful handling should be prioritized.

Table 1. Summary of Händel evaluation; runtime reported in seconds (s) and memory consumption in megabytes (MB)

	Phase III Time / Memory		
Benchmark	**AP1**	**AP2**	**AP3**
fop-events	14.79 / 343.5	36.03 / 996.3	37.2 / 976.3
fop-sandbox	1.4 / 21.3	104.78 / 2302.9	98.43 / 2062.5
fop-util	0.86 / 17.5	8.94 / 316.2	11.22 / 369.2
h2o-algos	0.42 / 14.7	0.4 / 17.4	0.39 / 14.4
h2o-avro-parser	0.6 / 25.2	0.68 / 36.1	0.76 / 40.8
h2o-clustering	0.42 / 14.7	0.56 / 36.7	0.65 / 39.9
h2o-genmodel	14.64 / 572.4	11.92 / 494	37.19 / 1331.7
h2o-hive	1.95 / 37.2	53.17 / 1809.5	45.45 / 1528.6
h2o-orc-parser	1.11 / 19.8	4.73 / 273.4	7.45 / 377.8
h2o-persist-drive	93.0 / 49.9	1.1 / 73.9	1.71 / 114.8
h2o-persist-gcs	0.81 / 28.2	1.2 / 60.9	1.61 / 82
h2o-persist-http	0.44 / 23.4	0.38 / 20.7	0.41 / 23.4
h2o-persist-s3	134.67 / 3165.7	19.9 / 494.1	143.41 / 3330.9
h2o-security	0.5 / 25.3	0.62 / 40.5	0.48 / 25.1
h2o-webserver-iface	0.5 / 23.9	0.41 / 14.8	0.5 / 24.2
sunflow-image	1.83 / 24.6	2.52 / 124.7	3.09 / 205.2
sunflow-system	1.09 / 19.2	2.14 / 196.2	2.34 / 229.7

4 Evaluation

We evaluate the efficacy of Händel by applying it to 17 benchmark programs. These benchmarks were selected from the dacapo-23.11-chopin benchmark suite [7]. Specifically, we analyzed a subset of packages from the fop, h2o, and sunflow projects. These packages ranged from 150-1600 LoC with an average of ~670 LoC. For each benchmark, we ran Händel to check for each anti-pattern and captured the time and memory consumption associated with each phase. Across the 17 benchmarks, Phase I averaged a runtime of 5.53 seconds and 323.6 MB of memory, and Phase II averaged a runtime of 36.8 seconds and 4501 MB of memory. Table 1 presents the runtime and memory of Phase III.

To study how Händel performs compared to existing approaches, we implemented two additional rules that mirror those provided by checkstyle 10.16.0 [11] and PMD 7.1.0 [34], two popular static Java bug finding tools [39]. Specifically, we implemented a rule that detects when a program catches an exception that is too generic and another that detects if a method propagates a generic exception. Although reminiscent of AP1 and AP2, these rules focus on the mere presence of generic exception types rather than analyzing control-flow sensitive subsumption relationships. These specifications utilize `generic_expression` as a weight predicate, which assigns exception types arbitrary weights. Suppose an audit permits catching `Exception`; in this scenario, an auditor can either remove the corresponding `generic_exception` fact from the database or deprioritize the result by setting the probability to a small value.

We applied our generic catch, generic throw, and their comparable rules in checkstyle and PMD to our benchmark programs, measuring each framework's identified violations, runtime, and memory consumption. The results between checkstyle and PMD contained small discrepancies due to varying definitions of a "generic" exception or "illegal" propagation. However, due to the flexible and modifiable definition of our weight predicate, Händel identified the same potential violations as the other frameworks individually. Figure 5 presents the runtime and memory consumption for each framework's generic catch detection. We observed comparable metrics for each framework's throw detection.

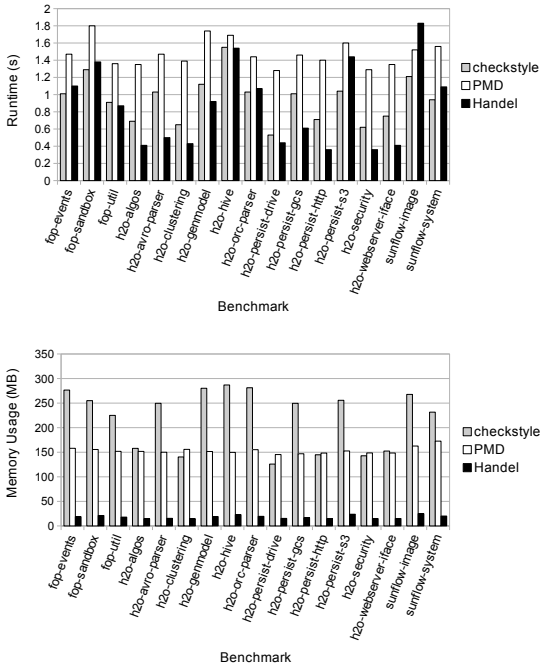

Figure 5. Runtime and memory performance for detecting a generic catch using Händel, checkstyle, and PMD

5 Discussion

Next, we discuss the implications of our preliminary evaluation, which aims to answer this question: Is probabilistic logic a suitable approach for detecting and reporting Java exception anti-patterns?

When evaluating an approach's suitability for a software auditing task, one must consider both usability and performance. The approach's programming interface should be amenable to easy expression and modification, while the resulting performance should be capable of efficiently accommodating the auditing needs. Our evaluation indicates Händel's promise in achieving both objectives, while future work will determine to what extent.

We evaluate Händel's usability by examining the expressiveness of its anti-pattern specifications. We report our findings based on our experiences constructing the ProbLog rules. Notice that the rule in Figure 4 is simple but logical. These

properties should make them amenable to comprehension and modification by software auditors. We intend further to study the expressiveness of Händel through usability studies. In terms of modification, we found that exposing Händel's configuration as ProbLog predicates presents an intuitive interface. Our design contrasts existing approaches, such as PMD or checkstyle, which utilize XML configuration files, a format designed for easier computer processing rather than human comprehension.

Often, expressiveness comes at the cost of performance. Therefore, an objective of our evaluation was to determine if the highly declarative detection logic of Händel would exhibit acceptable performance characteristics. Our benchmarks show promising performance and memory trends. The runtime of each anti-pattern varied across the evaluation benchmarks, with times ranging from less than half a second to 143 seconds, with an average of 15.9 seconds. We see a similar variation in memory consumption, which ranges from 14.4 to 3,224.4 MB, with an average of 427.7 MB. Although our benchmark programs are relatively small, the observed costs should be acceptable in most auditing scenarios. Furthermore, Händel is still in its prototyping phase, so we have not explored optimizations based on the underlying logic engine or methods to reduce the database size. We expect that applying such optimizations would strictly improve Händel's performance.

We also compared Händel with closely related tools concerned with finding defects in Java programs: checkstyle and PMD. Because these tools are not designed to detect control-flow-based defects, we introduced two additional specifications into Händel similar to closely related defect specifications in these existing tools. Across all evaluation benchmarks, Händel showed comparable or superior performance levels, which aligns with the well-known efficiency of logical inference. Furthermore, in Händel's case, the performance expenditure is a front-loaded one-time cost. Phases I and II, which are more expensive than existing approaches, only need to be executed once to establish the database of facts. All subsequent analyses can utilize the same database and benefit from the performance improvements.

6 Related Work

This work is related to Java anti-patterns, automated software auditing, and declarative program analysis. Prior works have defined catalogs of Java anti-patterns, including exception handling [5, 6, 12, 31, 43], dependency injection [27], concurrency [15], and performance [47] and demonstrated their presence in real software. Prior efforts have focused on creating tools to identify these anti-patterns [46, 54]. Our approach differs in specifying anti-patterns in a probabilistic logic language, thus providing weighted results.

Auditing and validating software is extremely resource-intensive, requiring significant money, developer effort, time, and expert knowledge [16, 32]. As a result, prior works have focused on automating the process [10, 30] or developing tools to identify software flaws [33]. In contrast to prior works that focus on identifying specific flaws, our work provides a more general framework for detecting software defects that can be expressed as logical rules.

Händel builds on extensive prior research on applying logic languages to solving program analysis problems. Logic languages are an effective means for specifying sophisticated and scalable analyses in a declarative manner [13, 21]. Past applications vary from calculating large-scale points-to relationships [8, 49, 50] to identifying structural program dependencies [18] or code property violations [45]. Händel's design takes inspiration from program analysis frameworks that express their analyses in a logic language [8, 23, 29]. The key novelty of our approach lies in employing *probabilistic* logic programming [14, 20, 25, 42] for specifying and executing program analyses.

7 Future Work and Conclusion

This paper presented a novel approach that facilitates auditing Java exception-handling logic. Our approach takes advantage of probabilistic reasoning and uses a logic language to declaratively express and configure auditing rules. We prototyped our approach in Händel. As a proof-of-concept, we revisited and specified three Java exception anti-patterns with Händel and demonstrated its ability to detect and prioritize potential anti-pattern matches.

Encouraged by our preliminary results, we plan to further study Händel's ability to specify auditing standards and detect violations. We envision our future work following three lines of inquiry. First, we plan to explore how extensible Händel is. To that end, Händel can be extended to support additional Java anti-patterns and anti-patterns in other languages. Second, we plan to explore Händel's performance and scalability. These characteristics are essential for Händel to provide practical benefits to software auditors. We can implement optimizations and expand our evaluation set to include larger code bases. Finally, we plan to explore Händel's usability further. We can conduct usability studies to understand whether Händel achieves the desired expressiveness and configurability.

As software reliability and quality remain an acute problem in software development, the need for approaches that facilitate auditing will only increase. By reporting on our experiences with Händel, we contribute novel designs and insights for using probabilistic reasoning in service of software auditing.

Acknowledgments

The authors thank the anonymous reviewers, whose insightful comments helped improve this paper. This research is supported by NSF through the grant #2232565.

References

[1] Martin Abadi and Luca Cardelli. 2012. *A theory of objects*. Springer Science & Business Media, New York, NY, USA.

[2] Amazon. 2024. *Amazon S3 examples using SDK for Java 2.x*. Retrieved May 1, 2024 from https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/java_s3_code_examples.html

[3] Amazon. 2024. *Developer Guide - AWS SDK for Java 2.X*. Retrieved May 1, 2024 from https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide

[4] Abdulbaki Aydin, Lucas Bang, and Tevfik Bultan. 2015. Automata-Based Model Counting for String Constraints. In *Computer Aided Verification (CAV '15)*, Daniel Kroening and Corina S. Păsăreanu (Eds.). Springer Cham, Cham, Switzerland, 255–272. https://doi.org/10.1007/978-3-319-21690-4_15

[5] Guilherme Bicalho de Pádua and Weiyi Shang. 2017. Studying the Prevalence of Exception Handling Anti-Patterns. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. IEEE, 328–331. https://doi.org/10.1109/ICPC.2017.1

[6] Guilherme Bicalho de Pádua and Weiyi Shang. 2018. Studying the relationship between exception handling practices and post-release defects. In *Proceedings of the 15th International Conference on Mining Software Repositories (MSR '18)*. ACM, New York, NY, USA, 564–575. https://doi.org/10.1145/3196398.3196435

[7] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, et al. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications (OOPSLA '06)*. ACM, New York, NY, USA, 169–190. https://doi.org/10.1145/1167473.1167488

[8] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly Declarative Specification of Sophisticated Points-to Analyses. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '09)*. ACM, New York, NY, USA, 243–262. https://doi.org/10.1145/1640089.1640108

[9] William J. Brown, Raphael C. Malveau, Hays W. McCormick III, and Thomas J. Mowbray. 1998. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley & Sons, Inc., New York, NY, USA.

[10] W. L. Bryan, S. G. Siegel, and G. L. Whiteleather. 1982. Auditing Throughout the Software Life Cycle: A Primer. *Computer* 15, 03 (March 1982), 57–67. https://doi.org/10.1109/MC.1982.1653973

[11] Checkstyle. 2024. *Checkstyle*. Retrieved May 26, 2024 from https://checkstyle.sourceforge.io

[12] Roberta Coelho, Jonathan Rocha, and Hugo Melo. 2018. A Catalogue of Java Exception Handling Bad Smells and Refactorings. In *Proceedings of the 25th International Conference on Pattern Languages of Programs (PLoP '18)*. The Hillside Group.

[13] Steven Dawson, Coimbatore R. Ramakrishnan, and David S. Warren. 1996. Practical Program Analysis Using General Purpose Logic Programming Systems—A Case Study. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation (PLDI '96)*. ACM, New York, NY, USA, 117–126. https://doi.org/10.1145/231379.231399

[14] Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. 2007. ProbLog: A Probabilistic Prolog and Its Application in Link Discovery. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI '07)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2462–2467.

[15] Mattias De Wael, Stefan Marr, and Tom Van Cutsem. 2014. Fork/Join Parallelism in the Wild: Documenting Patterns and Anti-Patterns in Java Programs Using the Fork/Join Framework. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ '14)*. ACM, New York, NY, USA, 39–50. https://doi.org/10.1145/2647508.2647511

[16] Elfriede Dustin, Thom Garrett, and Bernie Gauf. 2009. *Implementing Automated Software Testing: How to Save Time and Lower Costs While Raising Quality*. Pearson Education.

[17] Felipe Ebert, Fernando Castor, and Alexander Serebrenik. 2015. An Exploratory Study on Exception Handling Bugs in Java Programs. *Journal of Systems and Software* 106 (Aug. 2015), 82–101. https://doi.org/10.1016/j.jss.2015.04.066

[18] Michael Eichberg, Sven Kloppenburg, Karl Klose, and Mira Mezini. 2008. Defining and Continuous Checking of Structural Program Dependencies. In *Proceedings of the 30th international conference on Software engineering (ICSE '08)*. ACM, New York, NY, USA, 391–400. https://doi.org/10.1145/1368088.1368142

[19] Martin Fowler. 2019. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, Boston, MA, USA.

[20] Norbert Fuhr. 2000. Probabilistic Datalog: Implementing Logical Information Retrieval for Advanced Applications. *Journal of the American Society for Information Science* 51, 2 (2000), 95–110. https://doi.org/10.1002/(SICI)1097-4571(2000)51:2<95::AID-ASI2>3.0.CO;2-H

[21] Shan Shan Huang, Todd Jeffrey Green, and Boon Thau Loo. 2011. Datalog and Emerging Applications: An Interactive Tutorial. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data (SIGMOD '11)*. ACM, New York, NY, USA, 1213–1216. https://doi.org/10.1145/1989323.1989456

[22] Joern. 2024. *Joern: The Bug Hunter's Workbench*. Retrieved May 1, 2024 from https://joern.io

[23] Herbert Jordan, Bernhard Scholz, and Pavle Subotić. 2016. Soufflé: On Synthesis of Program Analyzers. In *Computer Aided Verification (CAV '16)*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer Cham, Cham, Switzerland, 422–430. https://doi.org/10.1007/978-3-319-41540-6_23

[24] Bill Joy, Guy Steele, James Gosling, and Gilad Bracha. 2000. The Java (TM) Language Specification.

[25] Angelika Kimmig, Bart Demoen, Luc De Raedt, Vítor Santos Costa, and Ricardo Rocha. 2011. On the Implementation of the Probabilistic Logic Programming Language ProbLog. *Theory and Practice of Logic Programming* 11, 2-3 (March 2011), 235–262. https://doi.org/10.1017/S1471068410000566

[26] Andrew Koenig and Bjarne Stroustrup. 1990. Exception Handling for C++. *Journal of Object-Oriented Programming* 3, 2 (1990), 137–171.

[27] Rodrigo Laigner, Marcos Kalinowski, Luiz Carvalho, Diogo Mendonça, and Alessandro Garcia. 2019. Towards a Catalog of Java Dependency Injection Anti-Patterns. In *Proceedings of the XXXIII Brazilian Symposium on Software Engineering (SBES '19)*. ACM, New York, NY, USA, 104–113. https://doi.org/10.1145/3350768.3350771

[28] David C. Luckham and W. Polak. 1980. Ada Exception Handling: An Axiomatic Approach. *ACM Transactions on Programming Languages and Systems* 2, 2 (April 1980), 225–233. https://doi.org/10.1145/357094.357100

[29] Mayur Naik. 2020. *Petablox: Large-Scale Software Analysis and Analytics Using Datalog*. Technical Report. Georgia Technology Research Institute, Atlanta, GA, USA.

[30] Anh Nguyen-Duc, Manh Viet Do, Quan Luong Hong, Kiem Nguyen Khac, and Anh Nguyen Quang. 2021. On the Adoption of Static Analysis for Software Security Assessment–A Case Study of an Open-Source e-Government Project. *Computers & Security* 111 (Dec. 2021), 102470. https://doi.org/10.1016/j.cose.2021.102470

[31] Ana Filipa Nogueira, José C. B. Ribeiro, and Mário A. Zenha-Rela. 2017. Trends on Empty Exception Handlers for Java Open Source Libraries. In *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER '17)*. IEEE, 412–416. https://doi.org/10.1109/SANER.2017.7884644

[32] Christian Payne. 2002. On the Security of Open Source Software. *Information Systems Journal* 12, 1 (2002), 61–78. https://doi.org/10.

1046/j.1365-2575.2002.00118.x

[33] Henning Perl, Sergej Dechand, Matthew Smith, Daniel Arp, Fabian Yamaguchi, Konrad Rieck, Sascha Fahl, and Yasemin Acar. 2015. VCCFinder: Finding Potential Vulnerabilities in Open-Source Projects to Assist Code Audits. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*. ACM, New York, NY, USA, 426–437. https://doi.org/10.1145/2810103.2813604

[34] PMD. 2024. *PMD: An extensible cross-language static code analyzer*. Retrieved May 26, 2024 from https://pmd.github.io/

[35] George Radin. 1978. The Early History and Characteristics of PL/I. *ACM SIGPLAN Notices* 13, 8 (Aug. 1978), 227–241. https://doi.org/10.1145/960118.808389

[36] Fabrizio Riguzzi and Terrance Swift. 2010. Tabling and Answer Subsumption for Reasoning on Logic Programs with Annotated Disjunctions. In *Technical Communications of the International Conference on Logic Programming, Leibniz International Proceedings in Informatics (LIPIcs, Vol. 7)*. Schloss Dagstuhl—Leibniz-Zentrum für Informatik, Saarbrüken, Germany, 162–171. https://doi.org/10.4230/LIPIcs.ICLP.2010.162

[37] Fabrizio Riguzzi and Terrance Swift. 2013. Well–definedness and efficient inference for probabilistic logic programming under the distribution semantics. *Theory and practice of logic programming* 13, 2 (2013), 279–302. https://doi.org/10.1017/S1471068411000664

[38] Martin P. Robillard and Gail C. Murphy. 1999. Analyzing Exception Flow in Java Programs. *ACM SIGSOFT Software Engineering Notes* 24, 6 (1999), 322–337. https://doi.org/10.1145/318774.319251

[39] Nick Rutar, Christian B. Almazan, and Jeffrey S. Foster. 2004. A Comparison of Bug Finding Tools for Java. In *15th International Symposium on Software Reliability Engineering*. IEEE, 245–256. https://doi.org/10.1109/ISSRE.2004.1

[40] Seemanta Saha, Mara Downing, Tegan Brennan, and Tevfik Bultan. 2022. PReach: A Heuristic for Probabilistic Reachability to Identify Hard to Reach Statements. In *Proceedings of the 44th International Conference on Software Engineering (ICSE '22)*. ACM, New York, NY, USA, 1706–1717. https://doi.org/10.1145/3510003.3510227

[41] Seemanta Saha, Laboni Sarker, Md Shafiuzzaman, Chaofan Shou, Albert Li, Ganesh Sankaran, and Tevfik Bultan. 2023. Rare Path Guided Fuzzing. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23)*. ACM, New York, NY, USA, 1295–1306. https://doi.org/10.1145/3597926.3598136

[42] Taisuke Sato and Yoshitaka Kameya. 2001. Parameter Learning of Logic Programs for Symbolic-Statistical Modeling. *Journal of Artificial Intelligence Research* 15 (2001), 391–454. https://doi.org/10.1613/jair.912

[43] Demóstenes Sena, Roberta Coelho, Uirá Kulesza, and Rodrigo Bonifácio. 2016. Understanding the Exception Handling Strategies of Java Libraries: An Empirical Study. In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR '16)*. ACM, New York, NY, USA, 212–222. https://doi.org/10.1145/2901739.2901757

[44] Saurabh Sinha, Alessandro Orso, and Mary Jean Harrold. 2004. Automated Support for Development, Maintenance, and Testing in the Presence of Implicit Flow Control. In *Proceedings of the 26th International Conference on Software Engineering (ICSE '04)*. IEEE, 336–345. https://doi.org/10.1109/ICSE.2004.1317456

[45] Leo St. Amour. 2017. *Interactive Synthesis of Code-Level Security Rules*. Master's thesis. Northeastern University, Boston, MA, USA. https://doi.org/10.17760/d20467254

[46] Ashish Sureka. 2016. Parichayana: An Eclipse Plugin for Detecting Exception Handling Anti-Patterns and Code Smells in Java Programs. (Dec. 2016). https://doi.org/10.48550/arXiv.1701.00108 arXiv:arXiv:1701.00108

[47] Catia Trubiani, Riccardo Pinciroli, Andrea Biaggi, and Francesca Arcelli Fontana. 2023. Automated Detection of Software Performance Antipatterns in Java-Based Applications. *IEEE Transactions on Software Engineering* 49, 4 (April 2023), 2873–2891. https://doi.org/10.1109/TSE.2023.3234321

[48] Bill Verners and Bruce Eckel. 2003. *The Trouble with Checked Exceptions: A Conversation with Anders Hejlsberg, Part II*. Retrieved May 15, 2024 from https://www.artima.com/articles/the-trouble-with-checked-exceptions

[49] John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. 2005. Using Datalog with Binary Decision Diagrams for Program Analysis. In *Programming Languages and Systems*. Springer Berlin Heidelberg, Berlin, Germany, 97–118. https://doi.org/10.1007/11575467_8

[50] John Whaley and Monica S. Lam. 2004. Cloning-Based Context-Sensitive Pointer Alias Analysis Using Binary Decision Diagrams. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI '04)*. ACM, New York, NY, USA, 131–144. https://doi.org/10.1145/996841.996859

[51] Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. 2012. Swi-prolog. *Theory and Practice of Logic Programming* 12, 1-2 (2012), 67–96. https://doi.org/10.1017/S1471068411000494

[52] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *2014 IEEE Symposium on Security and Privacy*. IEEE, 590–604. https://doi.org/10.1109/SP.2014.44

[53] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm. 2014. Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*. USENIX, 249–265.

[54] Lei Zhang, Yanchun Sun, Hui Song, Weihu Wang, and Gang Huang. 2012. Detecting Anti-Patterns in Java EE Runtime System Model. In *Proceedings of the Fourth Asia-Pacific Symposium on Internetware (Internetware '12)*. ACM, New York, NY, USA, 1–8. https://doi.org/10.1145/2430475.2430496

Received 2024-05-25; accepted 2024-06-24