Acta Cybernetica **27** (2025) 241–264.

Multithreading Atomicity: Static Analysis Checkers

Patrik P. Süli[a], Judit Knoll[b], and Zoltán Porkoláb[c]

Abstract

Ensuring thread safety in applications is crucial for preventing subtle and challenging bugs in concurrent programming. This paper presents two algorithmic approaches to improve thread safety through static analysis and to demonstrate their benefits in real life, the authors also implemented them as two detectors in SpotBugs static analyzer. These checkers are designed to identify unsafe usages of shared resources and improper atomic operations in concurrent Java programming, aiming to mitigate common multithreading issues such as race conditions. By emphasizing consistent locking strategies and the correct use of atomic types, the study offers insight into how to improve the reliability of multithreaded applications.

Keywords: Java, concurrency, atomic operations, static analysis

1 Introduction

Static analysis in software development can detect several types of issues, such as runtime errors and security violations in the code, without running the program itself, so developers could be informed about bugs in early stages of development [6]. There are many ways to analyze source codes, for example, Control Flow Analysis examines the execution, revealing infinite loops, unreachable codes, and improper usages of control structures [9]; Data Flow Analysis focuses on data tracking to identify issues like uninitialized variables, null pointer dereferences, and potential memory leaks [1]; Pattern-Based Analysis uses predefined rules to look for common issues or antipatterns in the code [13, 16].

Guidelines have been created to assist developers in producing code that is secure and reliable. The Software Engineering Institute (SEI) of the Carnegie Mellon University has its own, called CERT Coding Standards[1]. It has many rules

[a]Doctoral School of Applied Informatics and Applied Mathematics, Obuda University, Budapest, Hungary, E-mail: suli.patrik@uni-obuda.hu, ORCID: 0009-0001-9481-3664

[b]Sigma Technology Hungary Ltd., E-mail: judit.knoll@sigmatechnology.com, ORCID: 0009-0004-2400-6391

[c]Department of Programming Languages and Compilers, Institute of Computer Science, Faculty of Informatics, ELTE Eötvös Loránd University, Budapest, Hungary, E-mail: gsd@inf.elte.hu, ORCID: 0000-0001-6819-0224

[1]https://wiki.sei.cmu.edu/confluence/display/seccode/SEI+CERT+Coding+Standards

DOI: 10.14232/actacyb.311949

covering various aspects of coding practices, including memory handling, proper use of concurrency, input validation, and more, with the aim of preventing software vulnerabilities such as buffer overflows, race conditions, and injection attacks.

The SEI Cert Coding Standard contains several rules for atomicity, which is essential in software that work with parallel threads. This paper focuses on two specific rules of these, which are concerned with thread-safe usage of shared data.

There are several tools that can analyze code and make suggestions to improve it, one of them is SpotBugs[2]. SpotBugs is an open source tool that looks for possible issues in Java programs using Apache BCEL (Byte Code Engineering Library)[3], so it can handle binary *.class* files and understand instructions and methods at the bytecode level. When analyzing classes, SpotBugs reads and understands the structure of the bytecode, looking for specific patterns, coding practices, or known issues.

In this paper, we present two algorithms we designed and implemented as new detectors that cover possible concurrent programming problems in the Java language, which are described in the Visibility and Atomicity SEI Cert Rule group, focusing on the *VNA03-J*[4] and *VNA04-J*[5] rules. These rules provide practical guides with examples of both correct and incorrect usage, making it easy to identify common programming mistakes in connection with threads and shared resources, such as race condition, when a computation depends on timing or interleaving of multiple threads by the runtime. The *VNA03-J* rule highlights that a group of calls to independently atomic methods may not be atomic. *VNA04-J* underscores the method chaining mechanism, where the methods used in the chain can be atomic, but the chain overall is inherently non-atomic.

The rest of the paper is organized as follows: Section 2 introduces the concept of atomic types in different programming languages and details cases of non-thread-safe usage of this type. Section 3 presents the technical background used as a basis for our algorithms. The current state of the art is shown in Section 4 as a benchmark of a few static analyzer tools. The outline of the algorithms and the detectors developed are detailed in Section 5. The results are presented in Section 6, which are obtained from open source projects, with a comparison of their effectiveness with other static analysis tools. Furthermore, Section 7 highlights known limitations and opportunities for further development to improve the accuracy of the implemented detectors. The paper concludes in Section 8.

2 Related work

The concept of atomic types, also known as atomic operations or atomic classes, was introduced in concurrent programming to manage and manipulate shared data

[2]`https://spotbugs.github.io`

[3]`https://commons.apache.org/proper/commons-bcel/`

[4]`https://wiki.sei.cmu.edu/confluence/display/java/VNA03-J.+Do+not+assume+that+a+group+of+calls+to+independently+atomic+methods+is+atomic`

[5]`https://wiki.sei.cmu.edu/confluence/display/java/VNA04-J.+Ensure+that+calls+to+chained+methods+are+atomic`

safely and efficiently without the need for complex synchronization mechanisms. The term `atomic` in this context refers to operations that are completed as a single, indivisible, and unbreakable unit.

This idea was first proposed and explored in low-level hardware and assembly languages where atomic instructions such as "test-and-set" [3] or "compare-and-swap" (CAS) [11] were implemented directly by the CPU to facilitate safe concurrent access to shared memory.

2.1 Atomic types in programming languages

As multithreading and parallel processing have become more prevalent, the significance of atomic operations in high-level programming languages has grown. Although, this challenge is not unique: similar issues occur in a wide range of programming languages. Therefore, it is crucial to extend this analysis to different languages to gain a comprehensive understanding of how usable the atomic type is in different environments. By comparing the atomic types in Java (and other JVM based languages like Kotlin and Scala), C++, Python, and Rust, we can understand how different languages approach the challenge of concurrency and atomic operations, highlighting the strengths and trade-offs of each approach.

Java introduced `atomic` types with the release of Java 5 in 2004. Java's `java.util.concurrent.atomic` package[6] includes several atomic classes such as `AtomicInteger`, `AtomicLong`, `AtomicBoolean`, and `AtomicReference`. These classes leverage low-level atomic instructions to offer thread-safe operations on single variables without the overhead of locks[7]. For instance, `AtomicInteger` provides methods like `incrementAndGet()`, `decrementAndGet()`, and `compareAndSet()`, which are implemented to ensure that operations are completed without interruption.

Kotlin and Scala as JVM-based languages leverage the same concurrency mechanisms and atomic types provided by the Java platform. Developers can use the `java.util.concurrent.atomic` package directly and create atomic-typed classes in the same way as in Java.

```java
AtomicInteger atomicInteger = new AtomicInteger(0);
atomicInteger.incrementAndGet();
```

Code 1: Example usage of Java's `AtomicInteger` typed variable

One of the major high-level programming languages, the C++ programming language, was lack of proper solution for atomics until the C++11 standard defined the C++ memory model and introduced atomic classes[8] in the C++11 Standard Library [12]. The `std::atomic` template encapsulates the complexity of atomic instructions and provides a standardized interface for the developers.

[6] https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/package-summary.html

[7] https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/package-summary.html

[8] https://cplusplus.com/reference/atomic/atomic/

```cpp
std::atomic<int> counter{0};
counter.fetch_add(1);
```

Code 2: Example of C++ `std::atomic` variable

As C++ is a highly performance critical programming language, the instantiations of the `std::atomic` template, where the hardware provides atomic instructions handling the type parameter (e.g., `std::atomic<int>`) are compiled without any run-time overhead, while more complex template parameters, like structs have an external locking mechanism to provide atomicity.

Python is known for its simplicity and readability, making it a favorite for tasks ranging from web development to data science. However, Python's Global Interpreter Lock (GIL) presents challenges in concurrent programming[9]. To address atomic operations, Python relies on external libraries like *'atomicx'*[10] or built-in threading primitives[11] to simulate atomicity.

```python
from atomicx import AtomicInt

atomic_int = AtomicInt(0)
atomic_int.inc()
```

Code 3: Example of an atomic operation using Python's `atomicx` library

In contrast, Rust is designed with a strong emphasis on memory safety and concurrency without sacrificing performance. Rust's ownership system ensures memory safety, while its standard library provides built-in atomic types[12] like `AtomicBool`, `AtomicIsize`, and `AtomicUsize`. These types support thread-safe lock-free operations, making Rust an excellent choice for programming systems and applications that require high reliability.

```rust
use std::sync::atomic::{AtomicUsize, Ordering};

let atomic_usize = AtomicUsize::new(0);
atomic_usize.fetch_add(1, Ordering::SeqCst);
```

Code 4: Example of an `AtomicUsize` variable in Rust

Despite the differences in syntax, all these languages share a common foundation when it comes to atomic types. They provide atomic operations to manage shared resources in multithreaded environments. However, the same challenges persist across these, namely avoiding race conditions and managing the complexity of lock-free programming, so the algorithms that are detailed in this paper could be applicable for each programming language mentioned before.

[9] https://wiki.python.org/moin/GlobalInterpreterLock
[10] https://github.com/RuneBlaze/atomicx
[11] https://docs.python.org/3.10/library/threading.html
[12] https://doc.rust-lang.org/std/sync/atomic/index.html

2.2 Thread safety issues with Java concurrency types

When working with atomic types or synchronized collections in Java, it is important to understand that while individual method calls on these variables are atomic, combining these operations within a thread introduces potential thread safety issues. If an operation in a thread involves multiple atomic variables, proper synchronization is necessary to ensure that the entire operation remains atomic.

To bring attention to this issue, the *SEI Cert VNA03-J* rule – titled *"Do not assume that a group of calls to independently atomic methods is atomic"* – was created to avoid wrong usages of atomic typed variables and collections, which can lead to difficult-to-detect concurrency bugs.

```
private AtomicInteger a = new AtomicInteger(10);
private AtomicInteger b = new AtomicInteger(15);
// ...
a.get().add(b.get()); // Combination is not thread-safe
```

Code 5: Combine Java atomic typed variables unsafe

In code snippet 5, the `get()` method calls on both variables are atomic *separately*, but the `add()` operation itself is not thread safe, because meanwhile another thread may make changes on one or both variables, as it is shown on Figure 1.

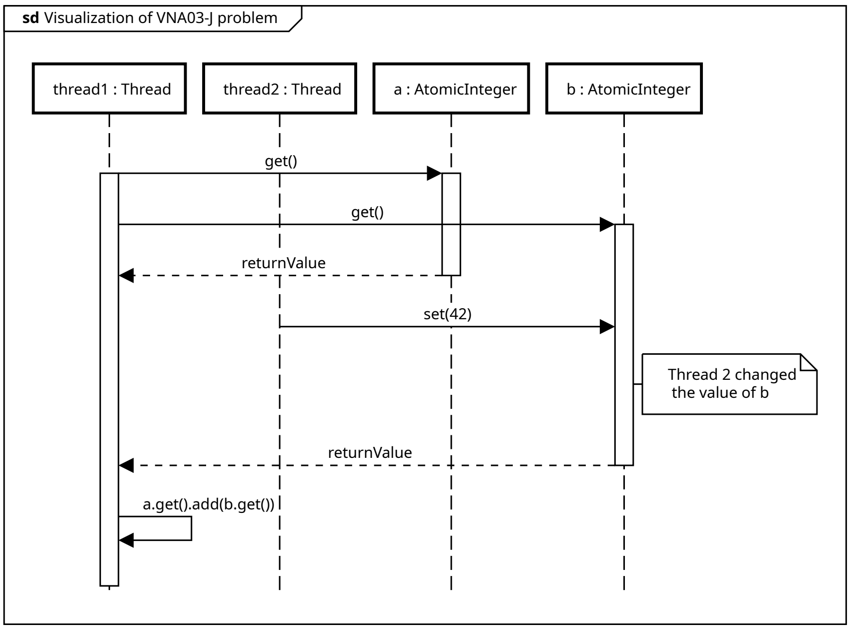

Figure 1: Sequence diagram of unsynchronized combination of atomic typed variables

2.3 Multiple atomic operations in threads

If a method contains more than one operation for an atomic variable without additional synchronization, then they are not atomic overall and could cause race condition between the threads, just like in section 2.2. So, to handle this, synchronization of code blocks or functions is necessary to guarantee that multiple threads cannot simultaneously modify or access shared resources.

```java
private AtomicInteger number = new AtomicInteger(0);
// Thread 1, combined atomic method calls are not atomic together
number.get();
// ...
number.get();
// Thread 2
number.getAndIncrement(); // It is safe standalone
```

Code 6: Unsafe multiple operations on atomic variable

`Thread 1` in Code 6 is not thread-safe, because between the two atomic method calls `Thread 2` can change the value of the variable.

2.4 Unsynchronized concurrent collection elements

The usage of thread-safe collections such as `SynchronizedList` or `ConcurrentHashMap` is not sufficient to ensure thread safety in itself, because any access to the collection's elements is not synchronized. The operations on these collections, such as adding or removing elements, are basically thread-safe, but accessing or modifying their elements themselves are not inherently synchronized. Consequently, any operations performed on the elements retrieved from these collections must be properly synchronized to avoid concurrent modification issues.

```java
private final Map<Integer, Integer> counterMap =
      Collections.synchronizedMap(new HashMap<Integer, Integer>());

public void incrementCounter(int id) { // Called by multiple threads
    Integer count = counterMap.get(id);
    counterMap.put(id, count + 1);
}
```

Code 7: Unsafe access to thread-safe collection elements

If multiple threads run the counter increment lines in Code 7 at the same time, it results in race condition, because the operation on the collection's element is not synchronized.

2.5 Unsafe usages of shared resources in multiple threads

The *SEI Cert's VNA04-J* rule - titled *"Ensure that calls to chained methods are atomic"* - is about the nonatomic-typed variable usages in threads. It focuses on a special case: method chaining.

Method chaining is a mechanism that allows multiple method calls on the same object in a single statement. It consists of a series of methods returning the `this` reference, allowing chained method invocations using the return value of the preceding method.

This style is often used in classes that employ the *Builder Pattern*[13] to set up objects with multiple parameters. For a common example, the `StringBuilder` class[14] uses this kind of mechanism.

```java
StringBuilder sb = new StringBuilder();
String result = sb.append("Hello, ").append("World!").toString();
```

Code 8: Example usage of `StringBuilder` class in Java

Although individual methods in a chain can be atomic, the chain as a whole is not. Consequently, callers must ensure sufficient locking for the chain's atomicity.

While the *VNA04-J* rule specifically deals with chained methods with builder pattern, its underlying principle can be extended to a wider range of scenarios. By generalizing this rule, a broader set of cases can be covered: any method can be found that modifies the state of a shared resource across multiple threads and may lead to race conditions. Overall, the checker is more flexible, it can recognize thread safety issues in various contexts, beyond just method chaining.

In Code 9 multiple threads modify the `User` object's state and when the execution reaches the `getName()` method call, the state of the `name` property is unambiguous.

```java
public class User {
    private String name;

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}

public class ExampleClient {
    private User user = new User();

    public ExampleClient() {
        new Thread(() -> {
            user.setName("Jane");
            System.out.println("New name: " + user.getName());
        }).start();

        new Thread(() -> {
            user.setName("Bob");
            System.out.println("New name: " + user.getName());
        }).start();
```

[13]https://refactoring.guru/design-patterns/builder

[14]https://docs.oracle.com/javase/8/docs/api/java/lang/StringBuilder.html

```
    }
}
```

Code 9: Unsafe operation in multiple threads to a shared resource

The visualisation of the problem in the *Code 9* can be seen in *Figure 2*.

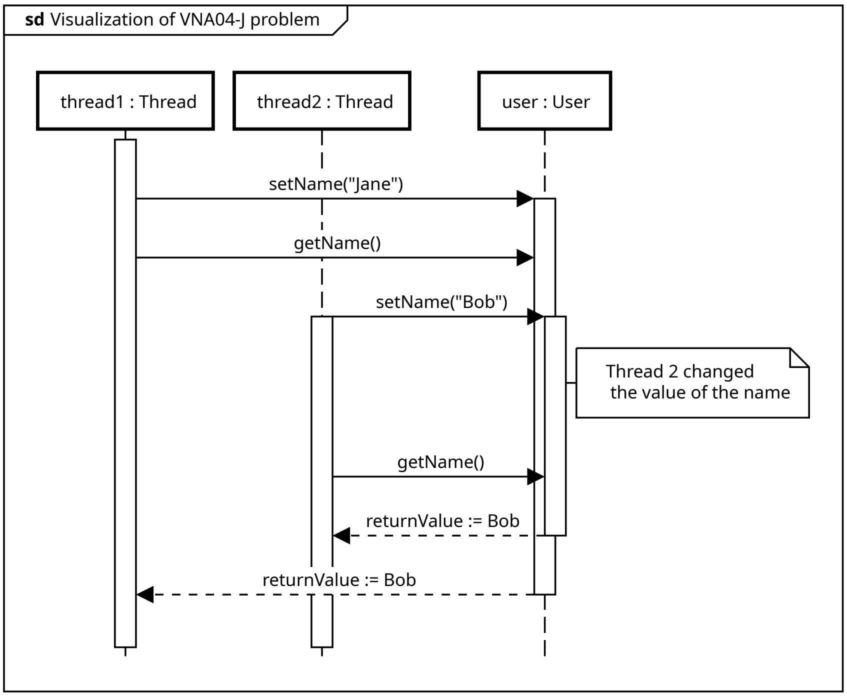

Figure 2: Sequence diagram visualization of Code 9

2.6 Similar concurrency issues in C/C++

Concurrency bugs involving atomic operations are not unique to Java. The SEI CERT guidelines for C and C++ also address atomicity and threading issues through rules such as *CON40-C* (*"Do not refer to an atomic variable twice in an expression"*)[15] and *CON43-C* (*"Do not allow data races in multithreaded code"*)[16].

These rules highlight problems analogous to *VNA03-J* and *VNA04-J* in Java, noting that while individual atomic operations (e.g., `atomic_load()`, `atomic_store()`, and compound assignments) are guaranteed to be thread-safe, combining multiple atomic reads or writes in a single expression or code block can result in race conditions without a careful locking mechanism. The recommendation in both languages

[15] https://wiki.sei.cmu.edu/confluence/display/c/CON40-C.+Do+not+refer+to+an+atomic+variable+twice+in+an+expression

[16] https://wiki.sei.cmu.edu/confluence/display/c/CON43-C.+Do+not+allow+data+races+in+multithreaded+code

to adopt explicit locking (e.g., using mutexes or synchronization) when performing compound operations.

Static analyzers in the C/C++ ecosystem, such as CodeSonar[17] or Coverity[18], provide support for detecting atomicity violations and data races according to the SEI Cert rules. Although each language's memory model differs in detail, the overarching principle remains the same: atomic types ensure thread safety only for *single* operations, and automated tools can help developers to detect and handle cases where multiple atomic operations compose a non-atomic sequence. By drawing these parallels, we emphasize that detecting improper atomic usages is a cross-language challenge that requires consistent locking strategies and thorough static analysis approaches.

3 Technical background

Contrary to testing and dynamic analysis methods, *static analysis* works at compile time, based only on the source code of the system, and does not require any input data [4]. Most of these methods are fast enough feasibly integrated into the continuous integration (CI) loop providing a positive impact on speed up the development-bug detection-bug fixing cycle. As the earlier a bug is detected, the lower is the cost of the fix [5], therefore, static analysis is a useful and relatively cheap supplement to testing.

All static methods apply heuristics, which means that sometimes they may *underestimate* or *overestimate* the program behavior [14]. In practice this means static analysis tools sometimes do not report existing issues which situation is called as *false negative*, and sometimes they report correct code erroneously as a problem, which is called as *false positive*. Therefore, all reports need to be reviewed by a professional who has to decide whether the report stands.

During the last two decades various static analysis techniques evolved. The most simple, but surprisingly strong method is *pattern matching*. First, the source code is transformed into some canonical format (e.g., all loops are converted to `while` and the body of the loop to a single line) and then predefined regular expressions are applied against this code. While context-sensitive problems (as divergence between the declaration and the use of a variable) are impossible to detect, many programmer's mistakes are detectable. As a huge advantage, this method does not require the successful construction of the Abstract Syntax Tree (AST), therefore applicable for non-compiling or partial code fragments too. Earlier versions of CppCheck[19] used pattern matching to find issues in C and C++ programs.

Most of the available static code analysis tools, however, are based on the analysis of the Abstract Syntax Tree (AST). The AST is a usual internal representation of a program or at least a translation unit used by the compiler [2]. Various versions of the AST can represent only the structure of the parsed tokens or may hold semantic

[17] `https://codesecure.com/our-products/codesonar/`
[18] `https://scan.coverity.com`
[19] `http://cppcheck.sourceforge.net/`

information too. It encodes the structure of the program, the declarations, the variable usages, selection and loop statements, function calls. Thus, AST-based static analysis is capable to detect complex errors, like erroneous implicit conversions, inconsistent design rules, and many others. Such checks are relatively fast, some of them may be implemented using a single traversal of the AST. These features make the AST-based method the most frequently used type of static analysis with notable examples as the Clang Tidy[20] for C++, SpotBugs for Java and PyLint[21] for Python.

While the AST-based method is more powerful than the regular expression based one, seeing only the *structure* of the program it still lacks of the reasoning on the *possible values* of the variables at a certain point of the program. *Symbolic execution* [10] is a path-sensitive abstract interpretation method. During symbolic execution we interpret the source code, but instead of using the exact (unknown) run-time values of the variables we use symbolic values and gradually build up constraints on their possible values. Symbolic execution is the most powerful, but also the most expensive method for static analysis, and requires a precise modeling of the language semantics and the representation of the memory usage [7].

3.1 SpotBugs overview

SpotBugs is designed to detect bugs in Java programs by analyzing bytecode. It is the successor to FindBugs and maintains compatibility with many of its features and plugins. SpotBugs can identify a wide range of potential issues in Java code, including but not limited to concurrency problems, performance bottlenecks, and potential security vulnerabilities.

SpotBugs primarily operates by analyzing the Abstract Syntax Tree (AST) generated from Java bytecode. It uses various detectors, which are specialized components designed to identify specific types of bugs. These detectors can be visitor-based, which analyze the bytecode in a straightforward manner, or CFG-based, which utilize control flow graphs to perform more sophisticated analysis. CFG-based detectors are particularly powerful, but come with higher computational costs.

One of the strengths of SpotBugs is its extensibility. Developers can create custom detectors through a plugin architecture, allowing SpotBugs to be tailored to specific project needs. The tool is capable of integrating into continuous integration (CI) pipelines, providing ongoing feedback on potential issues as code is developed.

3.1.1 Applying SpotBugs to Kotlin and Scala: FindSecBugs

SpotBugs, while originally designed for Java, can also be applied to other JVM-based languages like Kotlin and Scala. This is particularly useful in projects where multiple JVM languages are used, allowing for consistent static analysis across different parts of the codebase.

[20] `https://clang.llvm.org/extra/clang-tidy/`
[21] `http://pylint.pycqa.org/en/latest/`

To facilitate security-focused static analysis in these languages, the Find Security Bugs (FindSecBugs)[22] plugin extends SpotBugs' capabilities. FindSecBugs is a SpotBugs plugin that specializes in detecting security vulnerabilities in Java, Kotlin, and Scala code. Identifies potential security issues such as SQL injection, cross-site scripting (XSS), and improper validation of input data.

When applied to Kotlin and Scala, FindSecBugs leverages the underlying bytecode analysis capabilities of SpotBugs, adapting them to handle the syntactic and semantic differences of these languages. While Kotlin and Scala introduce language-specific constructs that may not map directly to Java, the bytecode they compile to is still compatible with SpotBugs' analysis techniques.

However, it is important to note that, due to differences in the way Kotlin and Scala handle certain programming concepts, such as lambdas and coroutines [8, 15], there may be limitations in the accuracy and coverage of the analysis. Despite this, FindSecBugs and the SpotBugs itself remain valuable tools for enhancing security in multi-language JVM projects, providing a unified approach to identifying and mitigating security risks across Java, Kotlin, and Scala codebases.

4 State of the art

With the help of test cases focusing on the above-mentioned issues which we developed for the SpotBugs testing framework, we performed a comparative analysis with other existing static analyzers for Java.

On our test cases (which are detailed in Section 6) there should be *22* hits on the *VNA03-J* cases and *10* hits on the *VNA04-J*, but it seems like the static analysis tools we tested do not detect these multithreading atomicity rules, as can be seen in Table 1.

Table 1: Result of static analyzer tools hits on *VNA03-J* and *VNA04-J* test cases

Name of the tool	VNA03-J hits	VNA04-J hits
PMD v7.0.0 [23]	0	0
SonarQube v9.9.5.90363 [24]	0	0
The Checker Framework v3.43.0[25]	0	0
Google's Error Prone v2.27.1[26]	0	0
SpotBugs v4.8.6[27]	0	0

Although all tools report atomicity-related issues, these are limited to other aspects, such as do not use the `volatile` keyword[28], and suggests replacing it with

[22]https://github.com/find-sec-bugs/find-sec-bugs

[27]This version of SpotBugs does not yet include the detectors detailed in this paper.

[28]https://docs.pmd-code.org/latest/pmd_rules_java_multithreading.html#avoidusingvolatile

a Java-built-in atomic type. These tools do not detect issues related to the complex usage of atomic types or synchronized collections[29].

In conclusion, our analysis indicates that the static analysis tools we tested do not currently detect the specific multithreading atomicity issues described in the *VNA03-J* and *VNA04-J* rules. SpotBugs is open source and free to use, it allows the detection of these bugs to be distributed to a large community of developers.

5 Detector algorithms

The *VNA03-J* and *VNA04-J* SEI Cert rules focus on the proper use of locking with synchronization as it is described in Section 2.2 and 2.5 in detail. We designed two algorithms and implemented them as detectors in the SpotBugs static analyzer to find unsafe usages of common references between threads, and make sure the proper usage of fields with Java `atomic` types. The algorithm that covers rule *VNA04-J* works with references of types which are not related to the Java Concurrent API, and the algorithm using the rule *VNA03-J* ensures the proper usage of `atomic` type-based classes.

The source code and test cases of these detectors are publicly accessible in the official SpotBugs repository, where our contributions are submitted as two pull requests: #2919 – *VNA03-J Sequence of calls on a synchronized abstraction may not be atomic*, and #2986 – *VNA04-J. Ensure that calls to chained methods are atomic*. VNA04-J is already available in SpotBugs from version 4.9.0.

5.1 Finding unsafe reference usages in multiple threads

The algorithm implementing *VNA04-J* rule works in class context, which means that it scans class bytecode, but it does not see the relations between classes and can only work with the code inside the currently analyzed class. This limitation is inherited from the SpotBugs Framework, as detailed in Section 7.2. The detector searches and collects methods that are in a call hierarchy that starts with a lambda (anonymous) or referenced method passed directly to a `Thread` object, but takes place in the current class. In Java, a `Thread` object[30] requires a method in its constructor that implements the `Runnable` functional interface.

We developed and tested a variation of the algorithm, in which the detector was designed to include all methods in its scan, meaning that it also works with the operations running on the main thread. When testing this solution on large open source projects (these are introduced in *Section 6*, with the final detection results) it had many false positive hits, so it was less useful on real world projects. Because of this we decided to use the stricter version of the algorithm, to only cover a subset of the original problem, but have more useful, accurate hits.

[29]The *VNA03-J* SEI Cert Rule Wiki page mentions that the Coverity and Parasoft Jtest (`https://www.parasoft.com/solutions/static-code-analysis/`) tools cover the rule, however, being proprietary tools and not freely available for research we do not cover them in our evaluation.

[30]`https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html`

Algorithm 1 Collect methods used in threads

```
1: $methodsInThread \leftarrow \emptyset$
2: for all method invokation bytecode instruction in methods do
3:     if method invokation implements "java.lang.Runnable" and is passed to
       "java.lang.Thread" then
4:         $methodsInThread \leftarrow invokedMethod$        ▷ This is a starting point
5:     else if CONTAINS($methodsInThread$, $currentMethod$) and
       $currentClass = invokedClass$ then
6:         $methodsInThread \leftarrow invokedMethod$
7:     end if
8: end for
```

When detecting the issue the class context is scanned twice, this is necessary, because the methods are visited in the order of definition, not in call order, so every method is visited once during one scan. The issue can only happen in methods which are used by threads, so in the first scan these relevant functions are gathered (as it is shown in Algorithm 1).

The second time the algorithm visits the code (see Algorithm 2), it looks for the usages of variables inside the stored methods. It collects the operations and groups them by variables which are performed on variables with not atomic or synchronized types.[31]

Every operation on the referenced fields - which meet the type constraint - is processed and the following boolean flags are saved about each variable:

- `onlySynchronized` is `true`, if all modifier operations are under proper synchronization.
- `onlyPutField` is `true`, if the threads only assign new values to the field.
- `modified` is `true`, if a thread modifies a variable or assigns a new value to it either directly or via a method call.

The `onlySynchronized` flag is necessary because, in the end, only those variables are relevant that have at least one operation not properly synchronized.

An acceptable solution could be that the threads only assign new values to fields and don't perform any other operations on them. E.g., construct the `message` variable with the help of `Builder` class, the construction of the object is finalized by calling the `build()` method. This pattern makes the `Message` class immutable and, consequently, thread-safe. The `onlyPutField` flag helps to identify this special case.

The `modified` flag is used to decide if there are multiple threads with only reading operations, since then it does not lead to race condition, but if at least one thread modifies the referenced object's state, then the state is not ambiguous in the threads.

[31] Improper usages of variables with Java's built-in atomic types and synchronized collections are handled by the VNA03-J algorithm.

A bug is reported, if a field is modified in multiple threads, accessed outside of synchronized blocks, and is neither a synchronized collection nor an atomic typed field. The algorithm marks instructions as a bug if shared data is used in multiple threads, with at least one modifying its state without a consistent locking policy.

The algorithm marks all instances of not thread-safe field accesses as a potential bug, because it helps the developer to identify which statements require synchronization, so in general it makes easy to locate and accurately determine the appropriate scope of the `synchronized` block necessary to ensure thread safety in a method.

Algorithm 2 Collect operation data in threads

```
1:  List $methodsInThread$                                  ▷ Inherited from Algorithm 1
2:  Map$\langle Field, FieldData \rangle$ $fieldInThreads \leftarrow \emptyset$
3:  for all bytecode instructions in $methodsInThread$ do
4:      if (field assignment or method call on field and
    CONTAINS($methodsInThread$, $currentMethod$)) then
5:          $data \leftarrow$ GETORCREATE($fieldsInThreads[field]$)
6:          $data.onlySynchronized \leftarrow data.onlySynchronized \wedge isSynchronized$
7:          $data.onlyPutField \leftarrow data.onlyPutField \wedge isFieldAssign$
8:          $data.modified \leftarrow data.modified \vee isFieldAssign \vee looksLikeSetter$
9:          $fieldsInThreads[field] \leftarrow$ PUTORUPDATE($data$)
10:     end if
11: end for
```

5.2 Finding non-atomic usages of the Java Concurrent types

The `atomic` typed fields and collections have atomic methods that are inherently atomic. The algorithm based on the rule *VNA03-J* searches for scenarios where these atomic methods are used in a combined or sequential manner. If a shared data is used more than once, the operations together are not atomic, so these accesses are marked as bug. There may be the possibility that all shared data are used just once in a method, but if combined (for example `a.get().add(b.get())`) then it is a bug too, because these two resource accesses must be atomic not only individually. It is important to note that, if a shared data is used by multiple methods and at least one accesses it more than once, all methods that work with it need synchronization for consistent locking to avoid race conditions between the threads that are using the same shared resource at the same time while parallel running.

The algorithm has the following base logic: analyzing functions in a class context and mark each method call and field assignment of common objects which are not `synchronized`. If a method contains a `synchronized` block, the detector logs only once for every different object inside the block, no matter how many times they are accessed; because of the synchronization, it is considered an atomic operation. If a private method performs an unsafe operation without proper synchronization, but all the methods that call it have proper synchronization, then the private function does not need another one.

Shared data could be a field of the class, a function argument, or a local variable containing a reference for a shared resource, for example, an element of an atomic collection.

The algorithm also visits the class two times: first, the `atomic` or synchronized collection typed fields of the class are collected. For fields with types inherited from the `atomic` package (such as `AtomicInteger`, `AtomicLong`, `AtomicBoolean`, and `AtomicReference`) only the type needs to be checked, but finding the synchronized collections is a bit more complex: the fields only has a general `List`, `Set` or `Map` type and the algorithm must look for the field assignments to determine the concrete type. For example, Code 10 shows a synchronized list assignment:

```java
List<String> lst = Collections.synchronizedList(new ArrayList<>());
```

Code 10: Create a synchronized list

Overall, a method which creates a synchronized collection can be recognized by being in the `Collections` class[32] (of the `java.util.concurrent.atomic` package), and its name starting with ”synchronized” followed by the concrete collection type’s name (e.g. `synchronizedSet`, `synchronizedMap`). The checker stores the variables, which are assigned the return value of these methods.

Algorithm 3 Check if a Class Member’s type is `atomic` or a synchronized collection

```
1: function IsAtomicTypedField($classMember$)
2:     $methodNames \leftarrow$ GetMethodNamesReturningSyncCollections()
3:     $className \leftarrow$ GetClassName($classMember$)
4:     $isCollectionsClass \leftarrow$ ”java.util.Collections” $= className$
5:     $isAtomicClass \leftarrow className$.StartsWith(”java.util.concurrent.atomic”)
6:     $isNameInteresting \leftarrow$ Contains($methodNames, className$)
7:     return $(isCollectionsClass \wedge isNameInteresting) \vee isAtomicClass$
8: end function
```

It is possible that a collection typed field has more than one assignment, and not all of them are synchronized collection assignments, for example, in the constructor a `List` collection is only assigned a simple `ArrayList` value, but after the application starts running, it is assigned a `synchronizedList` value. In this case, the checker treats this field as a synchronized collection, it assumes that the variable is used in concurrent operations.

In the second visit using the data of the collected variables, the detector looks for operations on these fields in every method in the class, except the constructor and synchronized methods. Constructors (as well as the static initializer) only run on object creation once, they do not appear in parallel operations, they may initialize fields, but this is part of the life-cycle of the object and can run only once. In addition to the stored fields, there may be local variables or method arguments with `atomic` types, and the checker also has to mark operations on these variables.

[32] `https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html`

Overall, the following operations are logged:

- A value assigned to a stored variable.
- A method called on a stored field.
- An operation on an `atomic` typed local variable or method argument.
- Multiple `atomic` typed variables combined.

When the algorithm has logged every instruction that meets the above list, it cumulates the results to determine which operations need to be reported as a bug. First of all, logged private methods are removed if they have proper synchronization on the call site in every method. Code 11 shows an example for this:

```java
private AtomicInteger count = new AtomicInteger(0);

public void modifyCountSafely() {
    synchronized (count) { // Every other caller methods call the
        private method with proper synchronization
        incrementAndPrint();
    }
}

private void incrementAndPrint() {
    count.incrementAndGet();
    System.out.println("Current count: " + count.get());
}
```

Code 11: Synchronized private method on the call side

This optimization only works with private methods because, with higher visibility, these methods can be accessed from outside the current class, and their usages are unknown (see Subsection 7.2).

After this, the algorithm has the information, which `atomic` typed fields are accessed multiple times by multiple methods without proper locking strategy. If operations are performed in multiple methods, these are marked as a bug, and it informs the developer to put these lines under a synchronization or refactor the usage strategy of the shared resource.

5.3 Generalization possibilities

Although our current algorithms are explained through the example of the Java language and contain language specific details, such as the `Atomic*` classes and standard library synchronized collections, our underlying detection logic can be extended to other programming languages, if they follow similar usage patterns (e.g. those mentioned in Section 2).

The core principle of identifying multiple potentially conflicting operations on shared data applies generally to any abstraction that offers atomic operations, but it can be composed unsafely if not synchronized consistently. However, in other programming languages, the same algorithmic idea remains valid, it must be adapted to detect their particular locking primitives – for example, `std::mutex` in C++, or `threading.Lock` in Python – instead of Java's `synchronized` blocks.

In SpotBugs, many Java-specific base type names are hardcoded rather than making them project-configurable. Since these elements are essentially part of the standard library, it is typically more practical and efficient to hardcode some parts of the detection logic than to parameterize it for every possible project. The synchronized collections used by the algorithms are specific to Java, but this approach could be extended to any language. For example, in C# the .NET Framework[33] offers `ConcurrentDictionary`, `ConcurrentQueue`, and `ConcurrentBag`, all of which ensure thread safety without requiring explicit external locking.

By enumerating these known synchronization and atomic constructs in other languages, the algorithms could be extended beyond Java to automatically detect non-atomic compositions of supposedly *atomic* operations.

6 Results

To validate our checkers, we implemented a considerable number of unit test cases to eliminate potential bugs and filter out possible false positive cases. After that, we evaluated our checkers on large, modern, open source Java projects, which were selected based on the following criteria:

1. **Concurrency intensity:** The project needs to use multithreading or concurrent data structures extensively.

2. **Codebase size and activity:** The project should be large and actively maintained, ensuring real-world relevance.

3. **Popularity and community participation:** The project should have a diverse contributor base and a significant user community, allowing meaningful feedback on potential bugs.

The unit tests are written in the SpotBugs testing framework, which makes them suitable to be used as integration test. Every test runs without issues, as expected.

For the *VNA03-J* there are *46* test cases overall: *22* positive and *24* negative to cover all possible mechanisms, such as edge cases like if there are multiple synchronized blocks in a method, but not every operation is inside, or lambda or anonymous method is passed to an atomic field's method call as argument, but this method itself performs additional operations on that same atomic field. Every test case has its own example class (which may have inner classes depending on the test's complexity) with a unique usage of atomic field(s).

For *VNA04-J* there are *14* test cases, with *10* positive and *4* negative. It has fewer test cases than the other checker, because in this case it is not necessary to include several atomic based types, it just works with any type that is not in Java's `atomic` package. However, it also includes some special cases like handling that if

[33] https://learn.microsoft.com/en-us/dotnet/standard/collections/thread-safe/

a `Thread` is passed to Java's `Runtime`[34] as a shutdown hook, or to verify that the checker can also work and detect bugs correctly with nested classes.

6.1 Results of detection the VNA03-J rule

The test results on the projects (can be seen in Table 2) show that the *VNA03-J* detector has low hit rate. We performed a *manual review* of each found bug by examining the relevant code regions, verifying whether the detected pattern could indeed lead to a race condition or data inconsistency. We confirmed that the reported issues were legitimate concurrency pitfalls. We found no code usage that was mistakenly classified as problematic, and we believe that *all the identified hits were true positive.*

Table 2: VNA03-J Measurements on large, open source projects

Project	Lines of Java Code	Atomic variables	Combined access bugs	Simple access bugs
Bt[35]	78 483	25	3	7
MATSim-Libs[36]	679 033	47	24	9
OpenGrok[37]	132 290	1	0	0
Kafka[38]	980 184	213	40	75
ElasticSearch[39]	3 149 220	339	71	99

Combined atomic accesses are reported where atomic variables are accessed multiple times in the same function without synchronization and marked cases of *simple atomic accesses*, when the access needs synchronization due of the existence of the combined resource usages in other methods.

Code 12 is a code snippet, a simplified version of the `Counter` class[40] originally from the MatSim-Labs open source repository, represents a real true positive finding. The class-level variable `nextCounter` is accessed multiple times – once with a `get()` call and again with a `compareAndSet()` call in the `incCounter()` method. These calls constitute an unsynchronized *combined atomic accesses* bug. Consequently, the `reset()` method, which also modifies `nextCounter`, can overlap with `incCounter()`, resulting in a *simple atomic access* bug.

[34] https://docs.oracle.com/javase/8/docs/api/java/lang/Runtime.html
[35] https://github.com/atomashpolskiy/bt/commit/6041303
[36] https://github.com/matsim-org/matsim-libs/commit/1c6779d
[37] https://github.com/oracle/opengrok/commit/077089f
[38] https://github.com/apache/kafka/commit/b436499
[39] https://github.com/elastic/elasticsearch/commit/44c92715
[40] https://github.com/matsim-org/matsim-libs/blob/1c6779d/matsim/src/main/java/org/matsim/core/utils/misc/Counter.java

```java
private final AtomicLong counter = new AtomicLong(0);
private final AtomicLong nextCounter = new AtomicLong(1);

public void incCounter() {
    long i = this.counter.incrementAndGet();
    long n = this.nextCounter.get();
    if ((i >= n) && (this.nextCounter.compareAndSet(n, n*multiplier)
        )) { // combined atomic access bug, multiple accesses
        log.info(this.prefix + n + this.suffix);
    }
}

public void reset() {
    this.counter.set(0);
    this.nextCounter.set(1); // simple atomic access bug
}
```

Code 12: Example of the relation between the bug types

While our research was more to find out the possibilities of detecting concurrency related errors with static analysis, we intend to apply our tool for solving practical problems. We initiated discussions with the maintainers of the projects where we find possible problems and we are looking for their feedback whether the findings were true positives. We hope a more intensive communication with theses developers when the new version of SpotBugs including our checkers will be available for the larger community.

6.2 Results of detection the VNA04-J rule

The test results for the *VNA04-J* rule on large, open source projects (with the same versions that are noted in Table 2) can be seen in Table 3. We found no hits on the evaluated projects because we opted to use the algorithm variation that excludes main thread analysis. This decision was made to avoid the checker being so noisy that it is unusable (see the details in Section 5.1).

Table 3: VNA04-J Measurements on large, open source projects

Project	Lines of Java Code	`Thread` starts	Unsafe access bugs
Bt	78 483	1	0
MATSim-Libs	679 033	1	0
OpenGrok	132 290	4	0
Kafka	980 184	1	0
ElasticSearch	3 149 220	0	0

It is very rare that in one class more than one `Threads` run parallel, which are using common resources. The detector has a serious limitation; it only works within the class context, and cannot see the relations, method calls outside of a class. This limitation is inherited from the SpotBugs framework, as discussed in Section 7. Is it possible that the public functions of a class are run in different threads in parallel way, but the checker cannot detect that usage.

6.3 Performance and integration with SpotBugs

SpotBugs keeps the analyzed application classes in memory (via BCEL) and performs separate passes on each classes for all enabled detectors, which are completely independent of each other, there is no shared data or any connection between them. This design simplifies the analysis process and avoids unintended interactions between detectors.

Our newly introduced concurrency checkers are implemented as additional detectors in SpotBugs. The *VNA03-J* algorithm, due to certain implementation details, is a CFG-Based detector, which means it is more expensive to run, than the visitor-based detectors, like the *VNA04-J*. To quantify runtime overhead, we ran SpotBugs on the *Kafka* codebase (same commit as noted in Table 2) on an Apple MacBook M2 Pro, repeating each run five times. We used the following command for benchmarking:

```
./gradlew clean core:spotbugsMain core:spotbugsTest -x test \
    --rerun-tasks --no-build-cache --profile
```

Table 4: Benchmark results on Kafka, run on an Apple MacBook M2 Pro

Configuration	**Avg. Runtime**
No concurrency detectors	15.21s
VNA03-J enabled	18.46s
VNA04-J enabled	15.54s
VNA03-J and VNA04-J enabled	18.79s

These results align with SpotBugs' documented tendency for CFG-based detectors to incur more overhead than visitor-based ones.

Similar to other SpotBugs detectors, our checkers rely on SpotBugs' in-memory class repository and do not add extra complex data structures. Consequently, we anticipate negligible memory overhead even when analyzing large-scale projects. Compared to existing detectors, our concurrency checkers primarily store per-class metadata for pattern matching and analysis and follow similar design principles and performance characteristics.

7 Known limitations and possibilities for further development

This section highlights limitations in the current implementation of the SpotBugs detectors and suggests some possible opportunities to improve them.

7.1 Use of atomic typed variables in public methods

Atomic types are typically used to ensure thread safety. However, if they appear in public methods not utilized in a threaded context, hits of the algorithm based on the rule *VNA03-J* are false positives. The algorithm assumes that, if the developer used atomic-typed variables, then it is because it is used in threads. This limitation could be solved by analyzing the relations of the classes (see Section 7.2), and exclude those public methods from the analysis which are not run in parallel way.

7.2 SpotBugs' class context analyzing limitation

SpotBugs operates within a single-class context, as such can only analyze operations within a class. This restriction may lead to potential false negatives in both detectors in systems where class interactions play a crucial role in the application's concurrency logic.

Without these limitations, the checker implementation of the algorithm based on the rule *VNA03-J* could exclude those nonprivate methods, which are not running in parallel threads, and the algorithm with rule *VNA04-J* could include those nonprivate methods which are passed to a `Thread` in another class, or just called by another method outside of the analyzed class that is running in parallel thread.

7.3 False positives in single-function modifications

Another issue is related to the scenario where multiple threads modify a common field, such as incrementing a counter. While this might technically represent a concurrency issue, if each thread's modification is self-contained and thread-safe (like atomic increments), it currently triggers a false positive.

```java
private AtomicInteger count = new AtomicInteger(0);

// Thread 1:
public void incrementByOne() {
    count.incrementAndGet();  // Safe atomic operation
}

// Thread 2:
public void incrementByTwo() {
    count.addAndGet(2);  // Another safe atomic operation
}
```

Code 13: Example of safe usage of parallel modifications

It is hard to determine by method names, what are those operations that only modify the common data, so since there were no hits like this in our evaluations of large open source projects, we chose to leave this false positive chance in the algorithm because it is not noisy for the developers.

7.4 Challenges with Lambda Expressions

The introduction of Lambda Expressions[41] into Java 8 marked a significant milestone in the evolution of the language. This feature was one of the most anticipated additions to Java and fundamentally changed the way Java programmers write code, especially when dealing with collections and concurrency.

Technically, a lambda expression in Java is an instance of a functional interface[42], an interface with a single abstract method (SAM interface). The Java compiler infers the type of lambda expression from the context in which it is used, allowing simpler and more concise syntax.

Lambda Expressions are implemented under the hood as bootstrap methods using the `invokedynamic` bytecode instruction. With the SpotBugs framework, there is some limitation to analyze lambda methods, because the calls and operations on these kinds of method are different due to the specialized bytecode instruction, and certain features are either not implemented or implemented in an alternative manner in the current version of SpotBugs, resulting in the loss of some information during analysis.

8 Conclusion

In concurrent programming, it is crucial to use shared resources in a thread-safe way. To achieve this, it is recommended to use a consistent locking policy, which could be even necessary, when a program works with Java `atomic` based types or with synchronized collections. Static analysis is a very useful tool to look for mistakes and make sure the developers identify and rectify potential errors, and implement their concurrent logics in a proper way.

We analyzed practices in thread safety, not only in Java but also by reviewing methodologies in other programming languages, such as Python, Rust and C++.

We delved into the SEI Cert Coding Standards, which is pivotal in guiding developers toward safer coding practices. Our research into this guideline was not just theoretical; we applied part of these standards practically by designing an algorithm and implementing corresponding checkers (which cover the *VNA03-J* and *VNA04-J* rules) in SpotBugs Static Analyzer Tool.

By integrating new detectors, our research has directly contributed to the enhancement of this tool, allowing it to identify unsafe resource usage across concurrent threads more effectively. The addition of these detectors extends SpotBugs' capa-

[41] https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html
[42] https://docs.oracle.com/javase/8/docs/api/java/lang/FunctionalInterface.html

bilities, enabling it to catch subtle bugs that could otherwise lead to inconsistent states or even system failures in production environments.

The enhancements in SpotBugs that we implemented offer practical benefits to developers by reducing the time and effort required to identify concurrency issues. This not only increases productivity, but also improves the overall reliability of software applications. By detecting potential problems in the early stages of the development cycle, developers can address issues before they manifest in deployed systems, reducing downtime and maintenance costs. These advantages also enable managers to reduce the use of project resources and financial expenditures.

Furthermore, our work underscores the value of community-driven open source projects in the evolution of software development tools. Our contributions to the SpotBugs project exemplify how individual efforts can lead to significant improvements in tools that are widely used by the developer community. The advanced capabilities of SpotBugs, enriched with more robust detectors for concurrency issues, render it a valuable tool for developers aiming to write safer and more reliable Java applications.

References

[1] Aghav, I., Tathe, V., Zajriya, A., and Emmanuel, M. Automated static data flow analysis. In *2013 Fourth International Conference on Computing, Communications and Networking Technologies (ICCCNT)*, pages 1–4, 2013. DOI: 10.1109/ICCCNT.2013.6726670.

[2] Aho, A. V., Sethi, R., and Ullman, J. D. *Compilers principles, techniques, and tools*. Addison-Wesley, Reading, MA, 1986. ISBN: 9780201100884.

[3] Anderson, T. The performance of spin lock alternatives for shared-money multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, 1990. DOI: 10.1109/71.80120.

[4] Bessey, A., Block, K., Chelf, B., Chou, A., Fulton, B., Hallem, S., Henri-Gros, C., Kamsky, A., McPeak, S., and Engler, D. A few billion lines of code later: Using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, 2010. DOI: 10.1145/1646353.1646374.

[5] Boehm, B. and Basili, V. R. Software defect reduction top 10 list. *Computer*, 34(1):135–137, 2001. DOI: 10.1109/2.962984.

[6] Bíró, P., Kádek, T., Kósa, M., and Pánovics, J. A new method to increase feedback for programming tasks during automatic evaluation. *Acta Polytechnica Hungarica*, 19(9):103–116, 2022. DOI: 10.12700/aph.19.9.2022.9.6.

[7] Clang SA Static Analyzer, 2019. URL: https://clang-analyzer.llvm.org/.

[8] Coroutines, K. O. D. https://kotlinlang.org/docs/coroutines-overview.html. Accessed: 08 2024.

[9] Halim, V. H. and Dwi Wardhana Asnar, Y. Static code analyzer for detecting web application vulnerability using control flow graphs. In *2019 International Conference on Data and Software Engineering (ICoDSE)*, pages 1–6, 2019. DOI: 10.1109/ICoDSE48700.2019.9092687.

[10] Hampapuram, H., Yang, Y., and Das, M. Symbolic path simulation in path-sensitive dataflow analysis. *SIGSOFT Software Engineering Notes*, 31(1):52–58, 2005. DOI: 10.1145/1108768.1108808.

[11] Herlihy, M. Wait-free synchronization. *ACM Transactions on Programming Languages Systems*, 13(1):124–149, 1991. DOI: 10.1145/114005.102808.

[12] ISO/IEC. N4917 post-summer 2022 C++. Working draft, International Organization for Standardization (ISO), Geneva, Switzerland, 2022. URL: https://isocpp.org/std/the-standard. Accessed: 07 2024.

[13] Palša, J., Hurtuk, J., Chovanec, M., and Chovancová, E. Using machine learning algorithms to detect malware by applying static and dynamic analysis methods. *Acta Polytechnica Hungarica*, 19(7):177–196, 2022. DOI: 10.12700/aph.19.7.2022.7.10.

[14] Rice, H. G. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74:358–366, 1953. DOI: https://doi.org/10.2307/1990888.

[15] Scala Coroutines. URL: https://scala-coroutines.github.io/coroutines/. Accessed: August 2024.

[16] Zhang, X., Zhou, Y., and Tan, S. H. Efficient pattern-based static analysis approach via regular-expression rules. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 132–143, 2023. DOI: 10.1109/SANER56733.2023.00022.