

TDK-thesis

Patrik Péter Süli

Multithreading Atomicity: Static Analysis checkers for Java and Comparative Language Insights

EÖTVÖS LORÁND UNIVERSITY

FACULTY OF INFORMATICS

DEPT. OF SOFTWARE TECHNOLOGY AND METHODOLOGY



Author:

Patrik Péter Süli

Computer Science MSc

IV. grade

Supervisors:

Judit Knoll

Senior Software Developer

Sigma Technology

Dr. Zoltán Porkoláb

Associate Professor

Eötvös Loránd University

Budapest, 2024

Contents

1	Introduction	3
2	Strategies to use resources thread-safe in different programming languages	5
2.1	C	5
2.1.1	Use Mutex Locks to guard the critical section of the code	5
2.1.2	Readers-Writer Locks to enhance read performance	6
2.1.3	Semaphores to coordinate fixed number of shared resources	7
2.2	C++	7
2.2.1	Resource Acquisition Is Initialization (RAII)	8
2.2.2	Atomic Operations	8
2.3	Java	9
2.3.1	Atomic access with the volatile keyword	9
2.3.2	Manage multiple threads with synchronization	9
2.3.3	Java built-in concurrency types	10
2.4	Perl	11
2.4.1	Sharing data structures between threads	11
2.4.2	Make shared data thread-safe with semaphores	11
2.5	Overview of the different approaches	12
3	Enhancing software quality and security through Coding Standards and Static Analysis	14
3.1	Coding Standards for enhanced software quality and security	14
3.2	Ensure Coding Standards through Static Analysis	17
4	Finding non-atomic operations with static analysis	20
4.1	Not thread-safe cases with Java concurrency types	20
4.1.1	Combination of atomic variables	21
4.1.2	Multiple atomic operations in threads	22

4.1.3	Unsynchronized concurrent collection elements	22
4.2	Unsafe usages of shared resources in multiple threads	23
4.3	Finding unsafe reference usages in multiple threads	25
4.4	Finding non-atomic usages of the Java Concurrent types	27
5	Results	30
5.1	Results of detection the VNA03-J rule	31
5.2	Results of detection the VNA04-J rule	32
5.3	Comparative analysis of existing static analyzer tools	33
5.4	Known limitations and possibilities for further development	34
5.4.1	Use of atomic typed variables in public methods	34
5.4.2	SpotBugs' class context analyzing limitation	35
5.4.3	False positives in single-function modifications	35
5.4.4	Challenges with Lambda Expressions	35
6	Conclusion	37
A	Unsafe Builder pattern usage with multiple threads	39
B	Compliant solution of using Builder pattern with multiple threads	41
	Bibliography	43
	List of Codes	48

Chapter 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. 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 (e.g., if, else, for, while) [1]; Data Flow Analysis focuses on data tracking to identify issues like uninitialized variables, null pointer dereferences, and potential memory leaks [2]; Pattern-Based Analysis uses predefined rules to look for common issues or antipatterns in the code [3].

Guidelines have been created to assist developers in producing code that is more secure and reliable. The Software Engineering Institute (SEI) of the Carnegie Mellon University has its own, called CERT Coding Standards [4]. It has many rules 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. These rules are applicable to different programming languages, and the SEI provides specific sets of rules for C, C++, Java, and other languages.

The SEI Cert Coding Standard contains several rules for atomicity, which is essential in softwares that work with parallel threads. This research focuses on two specific rules of these, which are concerned with thread-safe usage of shared data, and gives an insight and comparison of what solutions the programming languages have to write concurrent algorithms.

In Java, it is crucial to know, that the source code is not directly compiled into machine code. Instead, it is first transformed into platform-independent bytecode, that can be interpreted by the Java Virtual Machine (JVM), and this allows running the Java code on any device with a Java Runtime Environment (JRE) installed [5]. In addition, Java provides high-level concurrency constructs, such as synchronized blocks and methods, locking objects, and concurrent collections.

There are several tools that can analyze code and make suggestions to improve it, one of them is SpotBugs [6] (a fork of the abandoned FindBugs). It is an open source tool that looks for bugs in Java code using Apache BCEL (Byte Code Engineering Library) [7], so it can handle binary *.class* files and understand instructions and methods at the bytecode level. When analyzing classes, SpotBugs uses BCEL to read and understand the structure of the bytecode, looking for specific patterns, coding practices, or known issues.

In this research, I design and implement new detectors, I cover possible problems of concurrent programming in the Java language, which are based on the Visibility and Atomicity SEI Cert Rule group, focusing on the *VNA03-J* [8] and *VNA04-J* [9] rules. These rules have useful practical guides with compliant and non-compliant examples that helped to identify the individual cases of the programming mistakes that can be made with threads and shared resources, such as race condition, when a computation depends on timing or interleaving of multiple threads by the runtime. *VNA03-J* rule highlight that a group of calls to independently atomic methods may not be atomic. *VNA04-J* spotlight 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: *Chapter 2* discusses strategies to manage thread safety in programming languages and provides a comprehensive comparison of these strategies. *Chapter 3* explores coding standards that aim to improve software quality and security and how static analysis helps to comply these standards. After that, *Chapter 4* details the implementation of static analysis checkers that identify non-atomic operations in Java, providing an in-depth look at SEI Cert rules *VNA03-J* and *VNA04-J* and presents methodical approaches for detecting them in Java source code. The results are presented in *Chapter 5*, which are obtained from open-source projects, with a comparison of their effectiveness with other static analysis tools. Furthermore, it highlights known limitations and opportunities for further development to improve the accuracy of the implemented detectors. The paper concludes in *Chapter 6*.

Chapter 2

Strategies to use resources thread-safe in different programming languages

Effective management of resources in a concurrent programming environment is a fundamental challenge faced by developers in various programming languages. Ensuring thread safety, the condition of executing multiple threads simultaneously without leading to race conditions, deadlocks, or corrupting the state of shared resources, is paramount for the reliability and performance of software applications. Each of the widely used programming languages offers unique constructs, libraries, and paradigms to tackle concurrency, reflecting their individual philosophies and the evolution of programming practices over time.

2.1 C

Since C does not have built-in thread management like some higher-level languages, developers must use libraries such as POSIX Threads (pthreads) [10] for thread creation and control. In this library there are some ways to control shared resource accesses.

2.1.1 Use Mutex Locks to guard the critical section of the code

Mutex Locks (Mutual Exclusion) [11] are a foundational tool to synchronize access for shared resources. When a thread locks a mutex, other threads will not be able to lock the same mutex until it is unlocked, so it allows only one thread to access the locked resource at a time.

```
1 pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
2 pthread_mutex_lock(&lock); // Lock the mutex
3 // Access shared resources
4 pthread_mutex_unlock(&lock); // Unlock the mutex
```

Code 2.1: Use Mutex to guard a section of the code

Conditions [12] can be used to extend mutexes to synchronize access to shared resources. With it the different threads can communicate with each other with signals. One thread waits for a condition, while the other sends a signal, which makes the condition true.

```
1 pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
2 pthread_cond_t condition = PTHREAD_COND_INITIALIZER;
3
4 // Thread 1:
5 pthread_mutex_lock(&mutex);
6 while (!condition) {
7     pthread_cond_wait(&condition, &mutex);
8 }
9 // Access shared resources
10 pthread_mutex_unlock(&mutex);
11
12 // Thread 2:
13 pthread_mutex_lock(&mutex);
14 // Something that makes condition true
15 pthread_cond_signal(&condition);
16 pthread_mutex_unlock(&mutex);
```

Code 2.2: An example of using a condition variable in conjunction with mutex

2.1.2 Readers-Writer Locks to enhance read performance

When a concurrent algorithm involves high read and low write frequency to a shared resource, Readers-Writer lock [13] is more efficient than mutexes, because it can allow multiple access to read-only operations, but ensure exclusive access for write operations.

```
1 pthread_rwlock_t lock = PTHREAD_RWLOCK_INITIALIZER;
2 pthread_rwlock_rdlock(&lock); // Lock for reading
3 // Read from shared resource
4 pthread_rwlock_unlock(&lock);
5 // ...
6 pthread_rwlock_wrlock(&lock); // Lock for writing
7 // Write to shared resource
8 pthread_rwlock_unlock(&lock);
```

Code 2.3: Example of using Readers-Writer Locks

2.1.3 Semaphores to coordinate fixed number of shared resources

Semaphores [14] are a bit more flexible than mutexes, they maintain a count of how many units of a particular resource are available for a thread, which is useful for managing a fixed number of resources. Processes can increase (signal or release) or decrease (wait or acquire) the semaphore's counter.

```
1 #include <semaphore.h>
2 sem_t sem;
3
4 sem_init(&sem, 0, 1); // Initialize semaphore
5 sem_wait(&sem); // Wait and decrease semaphore
6 // Access shared resources
7 sem_post(&sem); // Signal and increase semaphore
```

Code 2.4: Example of using Semaphores

2.2 C++

C++ supports higher-level abstractions and language features that facilitate thread-safe access to shared resources, especially since C++11, where the mutexes and atomic operations have been added to the C++ standard library [15], that includes similar features from the C POSIX Threads library like threads (`std::thread`) [16], mutexes (`std::mutex`, `std::recursive_mutex`, etc.) [17], condition variables (`std::condition_variable`) [18], futures [19] and promises [20] for asynchronous programming.

2.2.1 Resource Acquisition Is Initialization (RAII)

The Resource Acquisition Is Initialization (RAII) programming technique [21] is used to manage resource lifecycles through the lifetime of objects. By tying resources to object lifetimes, C++ ensures that resources are automatically released when they are no longer needed, reducing the risk of leaks [22] or deadlocks.

```
1 std::mutex lock;
2
3 void example() {
4     std::lock_guard<std::mutex> lk(lock); // RAII class: mutex
5     // Access shared resources
6     if (!everything_ok()) {
7         return; // If the method returns normally, the mutex is
8                 // released
9     }
10 }
11 std::thread t(example);
12 t.join();
```

Code 2.5: Use RAII C++ programming technique [23]

2.2.2 Atomic Operations

The C++ standard library also includes support for atomic operations (`std::atomic`) [24] with a template class that represents types with atomic operations. This means that operations such as reading, writing, and updating the values of an atomic object can be performed without causing data races, even when these operations are performed from multiple threads.

```
1 std::atomic<int> counter{0};
2 // In threads:
3 counter++;
```

Code 2.6: Example of a `std::atomic` variable

2.3 Java

Java has a well-defined memory model [25] that specifies how threads interact through memory and what behaviors are allowed in concurrent execution. The memory model ensures visibility of changes to variables across threads under certain conditions, which can be enforced by using `volatile` keyword, synchronization blocks, or concurrent classes from the `java.util.concurrent` package.

Java's memory model helps prevent issues like memory visibility and ordering, making it easier to write predictable multithreaded programs. However, synchronized blocks or methods can significantly impact performance. Additionally, Java does not offer control over thread scheduling, which is handled by the JVM and garbage collection can introduce execution pauses that affect performance, especially in low-latency environments.

2.3.1 Atomic access with the `volatile` keyword

The `volatile` keyword [26] ensures that any thread that reads a field will see the most recent value written. It minimizes the risk of memory consistency errors, as any modification to a `volatile` variable sets up a happens-before relationship with subsequent reads of the same variable, so changes to `volatile` variables are always visible to other threads. This also implies that when a thread reads a `volatile` variable, it sees not only the latest change to the `volatile`, but also the side effects of the code that led up to the change.

```
1 private volatile int counter = 0;
2
3 public void increment() {
4     counter++;
5 }
```

Code 2.7: Atomic access of a `volatile` variable

This simple use of `volatile` ensures that any thread accessing `.counter` sees the most recent update made by any other thread.

2.3.2 Manage multiple threads with synchronization

The `synchronized` [27] keyword in Java is used to control access to a block of code or a method using multiple threads. When a method or block of code is declared with

synchronized, it ensures that only one thread can execute it at a time, thus preventing concurrent access issues such as race conditions.

```
1 class CommonResource {
2     private Integer number = 0;
3     public synchronized void update(int value) {
4         number = value;
5     }
6     public synchronized int addAndGet(int value) {
7         number += value;
8         return number;
9     }
10 }
11
12 private final CommonResource resource = new CommonResource();
13 // ...
14 new Thread(() -> resource.update(10)).start();
15 new Thread(() -> resource.addAndGet(5)).start();
```

Code 2.8: Using a resource thread-safe with the synchronized keyword

2.3.3 Java built-in concurrency types

The Java Concurrency API was introduced in Java 5, containing the `java.util.concurrent.atomic` package which includes classes that are useful in concurrent programming. [28] The `AtomicInteger`, `AtomicLong`, `AtomicBoolean`, and `AtomicReference` classes are created to perform atomic operations on single variables of integer, long, boolean and object reference types. In addition to this, the package contains concurrent collections that are thread-safe versions of standard Java collections (e.g. `ConcurrentHashMap` of `HashMap`).

```
1 private AtomicInteger number = new AtomicInteger(0);
2
3 public void update(int value) {
4     number.set(value);
5 }
6 public int addAndGet(int value) {
7     return number.addAndGet(value);
8 }
```

Code 2.9: Atomic usage of an `AtomicInteger` field

2.4 Perl

Perl employs a unique model known as "interpreter threads" (ithreads) [29], where threads do not share data unless explicitly specified. Shared variables must have to be marked explicitly, but just marking them is not enough to make them thread-safe. Like C and C++, Perl also works with manual locking mechanisms.

2.4.1 Sharing data structures between threads

By default, variables are exclusive to individual threads, and each new thread receives a private duplicate of every existing variable. The use of the `threads::shared` [30] module allows to share variables across different threads, which is essential for coordinating data between them. However, simply sharing does not ensure that access to these variables is thread-safe, so it is still necessary to handle synchronization manually.

```
1 my $counter :shared = 0;
2
3 sub increment_counter {
4     $counter++;
5 }
6
7 my $thread1 = threads->create(\&increment_counter);
8 my $thread2 = threads->create(\&increment_counter);
9
10 $thread1->join();
11 $thread2->join();
```

Code 2.10: Sharing a variable between threads in Perl

2.4.2 Make shared data thread-safe with semaphores

Semaphores are variables that are used to control access to some resource across multiple execution threads, ensuring that the correct number of threads can access a given part of the code at any time. In Perl's `Thread::Semaphore` [31] module maintains a counter that represents the number of permits available for accessing a particular resource.

```
1 my $sem = Thread::Semaphore->new(1); # Only one thread can enter
   at a time
2
3 sub critical_section {
4     $sem->down; # Acquire the semaphore
5     # Access shared resources
6     $sem->up;   # Release the semaphore
7 }
```

Code 2.11: Usage of semaphores in Perl

2.5 Overview of the different approaches

C, C++, Java, and Perl each approach the management of shared resources in multi-threaded environments differently, reflecting their unique design philosophies and capabilities.

In C, multithreading is handled through external libraries like POSIX threads, with developers directly managing shared resources using lower-level synchronization primitives such as mutexes and semaphores. This provides great flexibility and control, but requires careful attention to avoid issues like deadlocks and race conditions.

C++ builds on C's capabilities by integrating threading directly into the language with the C++11 standard, which includes thread management and synchronization features within the Standard Template Library (STL), though the STL itself is not an official part of the C++ library, but it is widely used. This approach improves safety and ease of use compared to C, while still offering fine-grained control over thread behavior and resource sharing.

Java offers a high-level approach to threading, deeply integrated into the language itself. Providing built-in thread management and a rich set of synchronization tools, such as synchronized blocks and volatile variables, to simplify safe access to shared resources. Java's standard library includes concurrent data structures that handle locking internally, making it easier to write safe and efficient multithreaded applications without deep knowledge of synchronization details.

Perl, with a less common use of threading, provides thread safety through its "interpreter threads" model, where data is not shared by default. The sharing between threads is managed explicitly using special modules such as `threads::shared`. This model

naturally avoids some typical concurrency problems, but can lead to higher memory consumption and requires a more deliberate effort to share state safely between threads.

In general, each language offers a distinct level of abstraction and integration for handling concurrency, with trade-offs between control, ease of use, and safety in the management of shared resources.

Chapter 3

Enhancing software quality and security through Coding Standards and Static Analysis

In the fast-evolving software development industry, ensuring the quality and security of code is paramount. The use of coding standards and static analysis in projects could help a lot in achieving high standards of software integrity and saving resource costs.

Coding standards serve as a cornerstone for developing robust, secure, and maintainable software, providing a set of guidelines that developers can follow to avoid common pitfalls and ensure consistency across the codebase. To augment these standards, static analysis emerges as an indispensable tool. Automates the evaluation of code against pre-defined standards, effectively scanning for deviations and potential vulnerabilities. By integrating static analysis into the development process, organizations can enforce coding standards more rigorously.

3.1 Coding Standards for enhanced software quality and security

Adoption of coding standards is highly recommended to ensure quality, security, and maintainability; the usage of them can improve projects in several ways [32]:

- **Consistency:** They promote uniformity in code, making it easier to understand, maintain, and manage.

- Security: By adhering to secure coding practices, developers can shield applications from common vulnerabilities.
- Safety: Ensures that safety-critical systems, such as those in the automotive, aerospace, and healthcare industries, meet strict regulatory requirements and operational safety. This reduces the risk of catastrophic failures due to software faults.
- Quality: Standards help detect code anomalies early in the development cycle, thereby improving overall quality.
- Efficiency: Well-defined standards streamline the development process, reducing the time and effort required for code reviews and testing.
- Team Collaboration: They simplify collaboration between team members by ensuring that everyone follows the same set of guidelines, which is particularly useful in large teams or teams distributed across different locations.

Some of the most popular coding standards are *MISRA C* [33], *OWASP Developer Guide* [34], *Google's Style Guides* [35] and the *SEI Cert Coding Standards* [4]. These standards provide developers with a comprehensive set of practices that are designed to prevent software errors, vulnerabilities, and coding failures.

MISRA C is a coding standard developed by the Motor Industry Software Reliability Association (MISRA) to enhance the safety, reliability, and maintainability of C code, particularly in the realm of embedded systems. This guideline is widely used across various industries such as aerospace, railway, and healthcare. The rules and directives focus on preventing common programming pitfalls, undefined behaviors, and potentially hazardous constructs. This ensures that safety-critical software adheres to the highest standards of quality and robustness, which is vital for systems where failure could have dire consequences. The latest version, *MISRA C:2023*, addresses modern C features and provides clearer and more comprehensive guidance to developers. Divide its recommendations into mandatory rules and advisory directives, allowing flexibility in adoption based on project requirements. [33]

The *Open Web Application Security Project (OWASP)* aims to empower developers, architects, and security professionals with the tools and knowledge to build secure web applications. OWASP provides open-source projects, documentation, and education initiatives that are freely available to the public. Its best-known resource is the OWASP Top Ten [36], which identifies the most critical security risks in web applications and offers

recommendations on mitigating these vulnerabilities. The Developer Guide complements the OWASP Top Ten by offering practical security concepts and essential details to help developers utilize various OWASP resources. It operates through a community-driven model, harnessing the expertise of volunteers who contribute to its wide range of security projects. [34]

Google's Style Guides maintain a set of style guides for many programming languages, including C++, Java, Python and JavaScript. These guides are designed to produce a readable and maintainable codebase, addressing aspects like naming conventions, file organization, comments, and usage of language-specific features. They also provide recommendations for testing, error handling, and performance optimization. Though created to maintain a high standard within Google's development teams, these guides are publicly available and widely adopted across the software industry. [35]

The *SEI Cert Coding Standards* are a set of secure coding guidelines developed by the Software Engineering Institute (SEI) at Carnegie Mellon University. Their primary objective is to assist software developers in producing secure, reliable, and compliant code. These standards offer recommendations for mitigating vulnerabilities, reducing the risks of software exploitation, and improving the overall quality of software systems. They are language-specific, providing tailored advice for programming languages such as C, C++, Java, and Perl. Each standard is organized around specific rules that address common programming errors, design flaws, or dangerous coding practices that can lead to security vulnerabilities. These rules cover areas such as input validation, memory management, concurrency, error handling, and object-oriented programming, and include detailed explanations of rationale, examples of both compliant and non-compliant code examples, and risk assessments that help developers prioritize the most critical issues. [4]

The *SEI CERT Oracle Coding Standard for Java* includes rules and recommendations. Rules are meant to provide normative requirements for code. Recommendations, on the other hand, offer guidance that should improve the safety, reliability, and security of software systems. These are divided into modules, which focus on different areas such as input validation, error handling, memory management, object-oriented programming, and visibility and atomicity. [37] *Chapter 4* aims at two rules from module visibility and atomicity and describes implemented detector solutions to cover these with static analysis.

3.2 Ensure Coding Standards through Static Analysis

In practice, to comply to coding standards is a multi-step process which may involve both manual review and static code analysis. These approaches ensure that developers consistently apply the standards and catch potential issues early in the development cycle.

Manual code reviews involve developers inspecting each other's code to ensure compliance with the defined coding standards. Moreover, it often leads to knowledge sharing and improved team cohesion. However, it can be time-consuming and prone to human error. In contrast, static code analysis is an automated solution that typically occurs at an earlier stage (at build time) in the development process. Static analysis tools automatically scan the source code to identify potential issues that violate coding standards or present well-defined risks. Unlike manual reviews, which rely on the subjective judgments of individual developers, static code analysis provides a systematic and consistent detection of common coding faults, which can enhance the efficiency of the development cycle.

Popular static analysis tools for Java are SonarQube [38], Coverity [39], PMD [40], The Checker Framework [41], Google's Error Prone [42] and SpotBugs [6], which are detailed in *Section 5.3*. The detectors of this research are implemented in SpotBugs, which is an open source static code analysis tool designed to examine Java code and identify potential bugs. It is a fork of the abandoned FindBugs project and continues its legacy by analyzing compiled Java bytecode to detect common bug patterns using the Apache Commons BCEL (Byte Code Engineering Library) library. [7]

The use of both manual reviews and static code analysis offers complementary benefits. Manual code reviews allow personalized feedback and mentorship, which can lead to better code understanding and personal development within the team. On the other hand, static code analysis offers a quick and systematic way to catch errors early in the development process.

In general, source code analysis involves examining and understanding a program's source code to uncover potential issues or identify specific issues that impact functionality, security, or performance. There are several approaches used in static analysis, each focusing on different aspects of the source code.

Control Flow Analysis [1] determines the sequence of executed instructions. It constructs a control flow graph (CFG), a visual representation showing the paths that control takes through the program. Each node in the CFG represents instructions, while edges are transitions between the blocks, based on control structures like loops and conditions

(if, else, for, while). It reveals key information about the program's structure, highlighting logical issues such as infinite loops, unreachable code (code blocks that never be executed), and improper usage of control structures. These are invaluable for optimizing compilers to improve execution speed and reducing overall software complexity.

```
1 private List<Integer> nums;  
2  
3 public void init() {  
4     nums = Collections.synchronizedList(new ArrayList<>());  
5 }
```

Code 3.1: An example code of a field assignment

For example, a CFG generated by SpotBugs can be seen in Figure 3.1 for the Code 3.1 snippet.

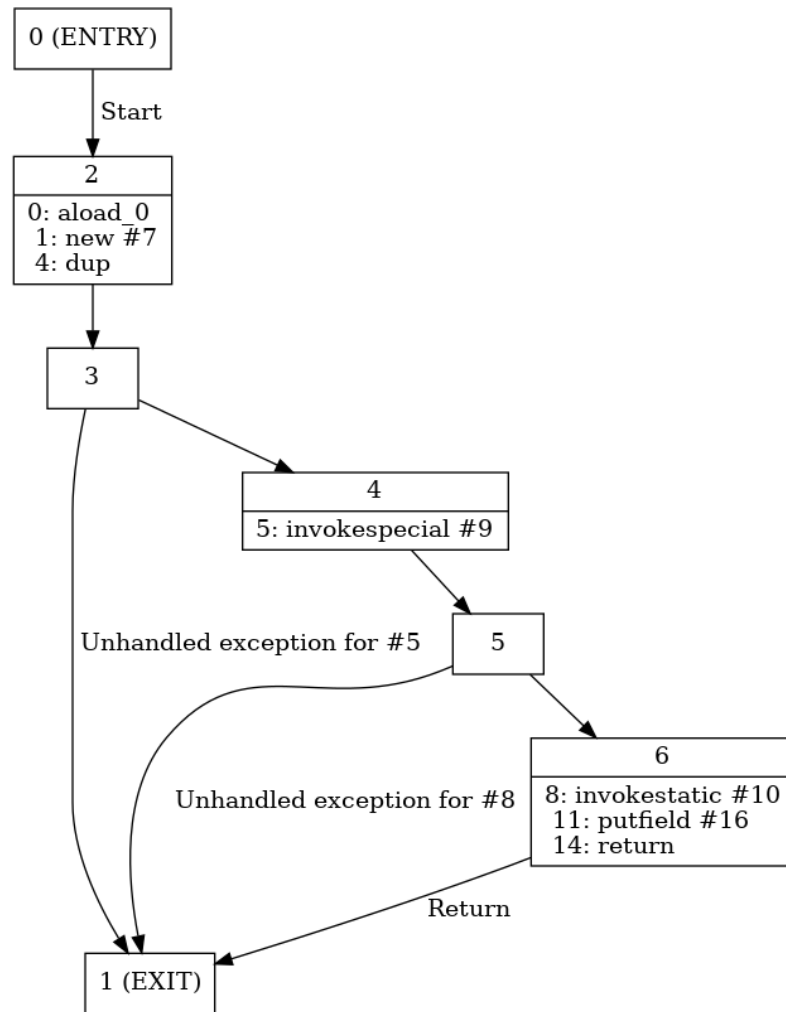


Figure 3.1: Example of visualized CFG generated by SpotBugs

Data Flow Analysis [2] examines how data values flow through a program to identify potential issues related to variable use and data integrity. By analyzing the paths that variables and data structures take throughout a program's execution, Data Flow Analysis can identify problematic coding practices like uninitialized variables (those that are used before being assigned a value), null pointer dereferences (accessing memory that hasn't been properly allocated), and potential memory leaks (allocated memory that is no longer referenced). This process creates a data flow graph that tracks assignments and uses of variables, which provides an overview of the data usage. The insights gained are essential in improving program reliability and security, because uninitialized variables and null pointer dereferences can cause unpredictable behavior or crashes, while memory leaks can lead to gradual performance degradation. Furthermore, complex software where variables interact intricately, such as in parallel computing or multi-threaded environments, can be better understood through Data Flow Analysis.

Pattern-Based Analysis [3] identifies common software issues by searching the source code for recognizable patterns known to lead to problematic behavior. These patterns (rules) can detect a range of problems like resource leaks, insecure coding practices, duplicated code, and naming inconsistencies. The rules are often based on known anti-patterns or coding standards, which guide developers to follow best practices. By comparing code against this rule set, the analysis flags potential vulnerabilities, performance bottlenecks, or violations of styling guidelines. Automating the detection of frequently encountered issues, reducing the risk of human error that can occur during manual code reviews.

Chapter 4

Finding non-atomic operations with static analysis

The *VNA03-J* and *VNA04-J* SEI Cert rules focus on the proper use of locking with synchronization as it is described in Section 4.1 and 4.2 in detail. I designed and implemented two checkers into the SpotBugs static analyzer to find unsafe usages of common references between threads, and make sure the proper usage of fields with Java atomic type. The checker that covers the rule *VNA04-J* works with references of which types are not related to the Java Concurrent API, and the other checker that implements the rule *VNA03-J* ensures the proper usage of atomic type-based classes.

4.1 Not thread-safe cases with Java concurrency types

As explained in section 2.3.3, the Java language has built-in concurrency types that provide methods to perform atomic operations on an encapsulated data structure (`AtomicReference`) or on a primitive type (`AtomicInteger`, `AtomicLong` or `AtomicBoolean`). Some of these atomic operation methods, for example `accumulateAndGet()`, do multiple operations, but overall these inner operations in this method are atomic, even if this method is called in different threads.

The purpose of the *SEI Cert VNA03-J* [8] rule - titled "Do not assume that a group of calls to independently atomic methods is atomic" - is to avoid wrong usages of atomic typed variables and collections.

4.1.1 Combination of atomic variables

The individual method calls on an atomic or synchronized collection-typed variables are atomic; however, if there is an operation in a thread which combines atomic variables, then synchronization is always necessary to ensure atomicity.

```

1 private AtomicInteger a = new AtomicInteger(10);
2 private AtomicInteger b = new AtomicInteger(15);
3 // ...
4 a.get().add(b.get()); // Combination is not thread-safe

```

Code 4.1: Combine Java atomic typed variables unsafe

In the code snippet 4.1, the `get()` method calls on both variables are atomic *separately*, but the `add()` operation is not thread safe, because meanwhile another thread may make changes on one or both variables, as it is shown on Figure 4.1.

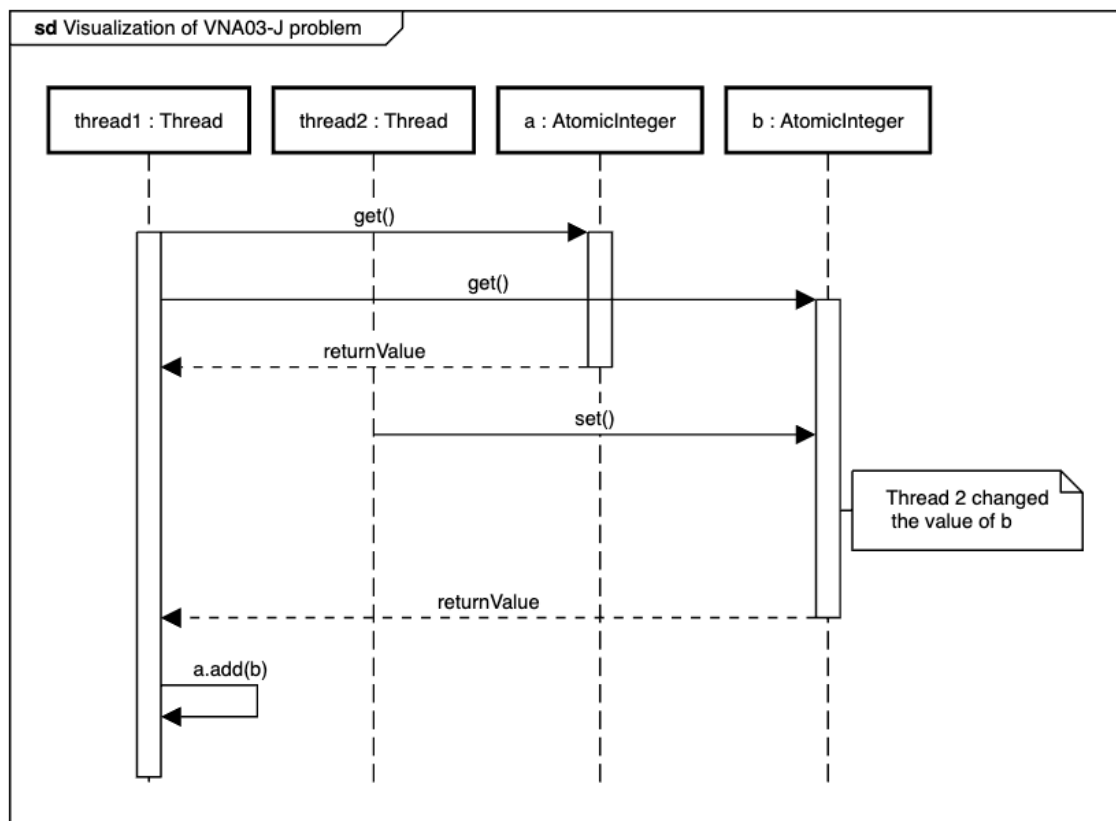


Figure 4.1: Example of unsynchronized combination issue of atomic typed variables

4.1.2 Multiple atomic operations in threads

Just like in section 4.1.1, when a method contains more than one operation for an atomic variable, these are not atomic overall and could cause race condition between the threads. So, to handle this, synchronization of code blocks or functions is necessary to guarantee that multiple threads cannot simultaneously modify or access shared resources.

```
1 private AtomicInteger number = new AtomicInteger(0);
2
3 // Thread 1, combined atomic method calls are not atomic together
4 number.get();
5 // ...
6 number.get();
7
8 // Thread 2
9 number.getAndIncrement(); // It is safe standalone
```

Code 4.2: Unsafe multiple operations on atomic variable

The Thread 1 in Code 4.2 is not thread-safe, because between the two atomic method calls, the second thread can change the value of the variable.

4.1.3 Unsynchronized concurrent collection elements

The usage of thread-safe collections such as `SynchronizedList` or `ConcurrentHashMap` is not sufficient, because any access to the collection's elements is not synchronized. The operations on these collections are basically synchronized, but reading or modifications on their elements are not, so any methods on these references may not be synchronized.

```
1 private final Map<Integer, Integer> counterMap =
2     Collections.synchronizedMap(new HashMap<Integer, Integer>());
3
4 // Multiple threads run the following code
5 Integer count = counterMap.get(id);
6 counterMap.put(id, currentCount + 1);
```

Code 4.3: Unsafe access to thread-safe collection elements

If multiple threads run the counter increment lines in of Code 4.3 at the same time, it results in race condition, because the operation on the collection's element is not synchronized.

4.2 Unsafe usages of shared resources in multiple threads

The *SEI Cert's VNA04-J* rule [9] - 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* [43] to set up objects with multiple parameters. For a common example, the `StringBuilder` class [44] uses this kind of mechanism.

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

Code 4.4: Example usage of `StringBuilder` class in Java

Although individual methods in a chain can be atomic, the chain itself is not. Consequently, callers must ensure sufficient locking for the chain's atomicity. The example class and client code in *Appendix A* demonstrates an unsafe usage of builder class in multiple threads.

This rule is too specific, because it only deals with chained methods with builder pattern. However, it can be generalized by stating that any method that changes the state of a shared resource used in multiple threads at the same time, can lead to race condition.

In Code 4.5 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.

```
1 public class User {  
2     private String name;  
3  
4     public void setName(String name) {  
5         this.name = name;  
6     }  
7  
8     public String getName() {  
9         return name;  
10    }  
11 }  
12  
13
```

```

14 public class ExampleClient {
15     private User user = new User();
16
17     public ExampleClient() {
18         new Thread(() -> {
19             user.setName("Jane");
20             System.out.println("New name: " + user.getName());
21         }).start();
22
23         new Thread(() -> {
24             user.setName("Bob");
25             System.out.println("New name: " + user.getName());
26         }).start();
27     }
28 }

```

Code 4.5: Unsafe operation in multiple threads to a shared resource

The visualisation of the problem in the *Code 4.5* can be seen in the *Figure 4.2*.

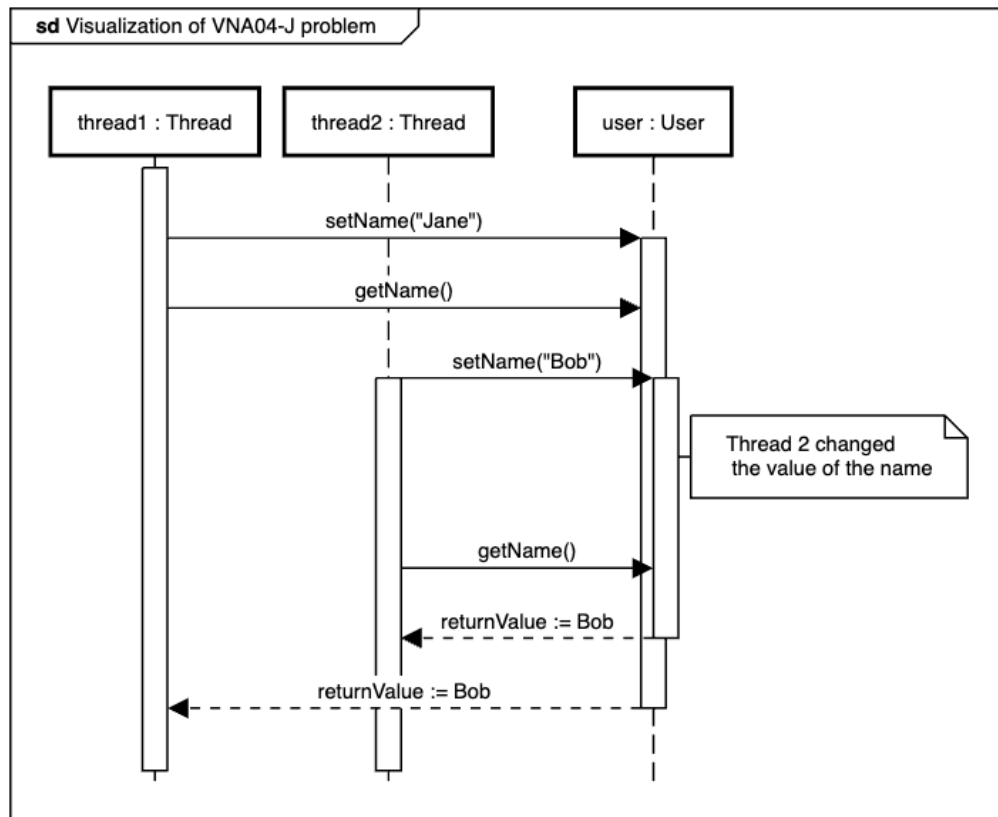


Figure 4.2: Sequence diagram visualization of Code 4.5

4.3 Finding unsafe reference usages in multiple threads

The checker which implements *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. It scans a context twice; the first time it 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 [45] requires a method in its constructor that implements the `Runnable` functional interface.

In the first phases of development the detector includes all methods to scan, so it means that the detector works also with the operations running on the main thread. I tested this solution on large open source projects (these are introduced in *Chapter 5*, with the final detection results), and it had many false positive hits, so it was less useful on real projects. Because of this I decided to make the checker stricter, to only work with parallel threads in a class.

Algorithm 1 Collect methods used in threads

```

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

```

It is necessary to scan a context twice, because the methods in a class's bytecode is not in call order, because *SpotBugs* scans the methods by definition order and analyzes every method only once, so in the first phase the checker needs to collect all methods to store those methods which are used by threads.

The second time the checker starts scanning the code again, but this time looks for the usages of variables inside the stored methods. It collects variable operations grouped by variables if the variable's type is not Java's atomic typed or is not a synchronized collection.¹

¹Proper usage of variables with Java's built-in atomic types is scanned by the *VNA03-J* detector

Every operations on the referenced fields which meets the type constraint is logged in the detector. The log contains the following boolean flags about the variable:

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

To avoid race conditions, proper synchronization is must have, so the checker in the end counts only those operations which are not under the proper locking mechanism. This is the reason why `onlySynchronized` flag is necessary.

An acceptable solution could be that the threads only assign new values to fields and do not do any other operations on them. Appendix B shows an example of using Builder pattern in compliant. All threads are 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 also necessary because if there are multiple threads with only reading operations, then it does not lead to race condition, but if one of the threads modified the referenced object's state, then the state is not ambiguous in the threads.

In the end a bug is reported, if a field is modified in multiple threads (except if these modifications are only field assignments), it is not only accessed in synchronized blocks, and it is not a synchronized collection or an atomic typed field. In summary, if the detector finds shared data that is used at least once in multiple threads and one of them changes its state, and overall a consistent locking policy is not used to every statement, it marks the instructions as a bug.

The detector marks all instructions as a bug because, as a developer, it is easy to find which statements should be under synchronization, so in general it is easier to locate the group of non-atomic operations, and to specify the size of the synchronized block.

Algorithm 2 Collect operation data in threads

```

1: List methodsUsedInThread ▷ Inherited from the Algorithm 1
2: Map $\langle$ Field,FieldData $\rangle$  fieldDataUsedInThreads  $\leftarrow \emptyset$ 
3: for all bytecode instructions in methodsUsedInThread do
4:   if (field assignment or field method call and CONTAINS(methodsUsedInThread,
     currentMethod) then
5:     data  $\leftarrow$  GETORCREATE(fieldsUsedInThreads[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:     fieldsUsedInThreads[field]  $\leftarrow$  PUTORUPDATE(fieldsUsedInThreads[field], data)
10:   end if
11: end for

```

4.4 Finding non-atomic usages of the Java Concurrent types

The atomic typed fields and collections have atomic methods that are inherently atomic, so the *VNA03-J* checker specifically 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 are not atomic, so these accesses are be 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. 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 checker has the following base logic: analyzing functions in a class context and log 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 detector also has two cycles: first, it collects the atomic or synchronized col-

lection typed fields in a class and stores them. The `AtomicInteger`, `AtomicLong`, `AtomicBoolean`, and `AtomicReference` typed fields are clear, that they are inherited from the `atomic` package by its type, but the synchronized collections not, because the fields only has a general `List`, `Set` or `Map` type and the checker must look for the field assignments to determine the concrete type. For example, a synchronized list assignment as seen in Code 4.6:

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

Code 4.6: Create a synchronized list

Overall, a method which creates a synchronized collection can be recognized by being in the `Collections` class (of the `java.util.concurrent.atomic` package [46]), and having its name the "synchronized" prefix followed by the concrete collection type's name. The checker stores those variables, which are assigned the result of these methods.

Algorithm 3 Check if a Class Member is atomic typed or a synchronized collection

```

1: function ISATOMIC_TYPED_FIELD(classMember)
2:   collectionNames  $\leftarrow$  {"synchronizedCollection", "synchronizedSet",
3:     "synchronizedSortedSet", "synchronizedNavigableSet", "synchronizedList",
4:     "synchronizedMap", "synchronizedSortedMap", "synchronizedNavigableMap"}
5:   className  $\leftarrow$  GET_CLASS_NAME(classMember)
6:   isCollectionsClass  $\leftarrow$  "java.util.Collections" = className
7:   isAtomicClass  $\leftarrow$  className.STARTSWITH("java.util.concurrent.atomic")
8:   isNameInteresting  $\leftarrow$  CONTAINS(collectionNames, className)
9:   return (isCollectionsClass  $\wedge$  isNameInteresting)  $\vee$  isAtomicClass
10: end function

```

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

In the second cycle with the data of the collected variables, the detector looks for operations on these fields in every method in the class context, except the constructor and those methods that are synchronized on the method level. Constructors (as well the static initializer) only run on object creation once, they do not appear in parallel operations, they may initialize fields, but this life-cycle cannot be run on multiple times because of its behavior. In addition to the stored fields, there may be local variables or method arguments

of which the types are also atomic, and the checker also has to log operations on these kinds of variables.

Overall, these are the operations, which have to be logged:

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

When the checker has logged every instruction that meets the above list, it cumulates the results to determine what operations need to be reported as a bug. First of all, to make the detector less noisy, it makes an optimization on the results. Remove all logged private methods that have proper synchronization on the call site in every method. The following code snippet shows an example for this:

```
1 private AtomicInteger count = new AtomicInteger(0);
2 public void modifyCountSafely() {
3     synchronized (count) { // Every other caller methods call the
4         incrementAndPrint() with proper synchronization
5         incrementAndPrint();
6     }
7 }
8 private void incrementAndPrint() {
9     count.incrementAndGet();
10    System.out.println("Current count: " + count.get());
11 }
```

Code 4.7: Synchronized private method on the call side

This optimization only works with private methods because, with more opened visibility, these methods can be accessed from outside the current class context, and their usages is unknown.

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

Chapter 5

Results

To validate my checkers, first I implemented a large number of unit test cases to eliminate potential bugs and filter out possible false positive cases. After that I evaluated my checkers on large, modern, open source Java projects, which applied concurrent solutions to get real-world measurement data.

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, in which there are 22 positive and 24 negative to cover all possible mechanisms, such as special ones like if there are multiple synchronized blocks in a method, but not every operation put there, or lambda or anonymous method is passed to an atomic field's method call as argument, but this method uses the atomic field itself again for other operations. 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 a `Thread` is passed to Java's `Runtime` as a shutdown hook [47], or the checker can also work and detect bugs correctly with nested classes.

For real-world measurement, I used the following open-source projects:

Bt is a Java library to create BitTorrent clients using a framework. It allows developers to build applications that can download and share files using the BitTorrent protocol. Bt provides an API that supports various networking strategies, custom message handling, and more. [48]

MATSim (Multi-Agent Transport Simulation) is implementing large-scale agent-based transport simulations. It provides tools to model transport systems and analyze how changes in infrastructure, policies, or behaviors affect traffic patterns. MATSim is widely used for research and planning by transport engineers and planners. [49]

Jenkins is one of the most popular open-source automation server that facilitates continuous integration and continuous delivery (CI/CD) in software development. It helps automate the parts of software development related to building, testing, and deploying, enabling developers to integrate changes to the project more quickly and with fewer errors. [50]

OpenGrok is a powerful search and indexing engine designed for source code. Provides fast and searchable access to source code repositories in a web-based interface. It is particularly useful for developers who need to understand large codebases or historical changes. OpenGrok features a full text search engine and can integrate with popular version control systems such as Git, Mercurial, and SVN. [51]

Torrent is a simple Java-coded BitTorrent client, intended to be used as a library for developing applications that require BitTorrent functionality. It can be embedded into other applications or used as a standalone program. Ttorrent supports downloading, seeding, and creating torrents, making it a versatile tool for developers looking to implement the capabilities of the BitTorrent protocol in their software. [52]

5.1 Results of detection the VNA03-J rule

The test results on the large, open-source projects (can be seen in Table 5.1) show that the *VNA03-J* detector has low hit rate, so it is not noisy. The marked errors are confirmed by manual review process that I conducted, ensuring the prevention of false-positive alerts. In the final state, I found no false-positives, all of the hits were true positive.

I marked *combined atomic accesses* issues where atomic variables are accessed multiple times in the same function without synchronization and marked cases of *simple atomic*

Project	Lines of Java Code	Count of combined access bugs	Count of simple access bugs
Bt	80714	6	14
MATSim-Libs	154975	48	18
Jenkins	311151	0	0
OpenGrok	146324	0	0
Ttorrent	21420	0	0

Table 5.1: VNA03-J Measurements on large, open source projects

accesses, when the access needs synchronization due of the existence of the combined resource usages in other methods.

```

1 private AtomicInteger num = new AtomicInteger(0);
2 public void conditionalUpdate(int val) {
3     if (num.get() < 0) {
4         num.set(val); // combined atomic access bug, multiple
                        // access of var
5     }
6 }
7 public int addAndGet(int val) {
8     return num.addAndGet(val); // simple atomic access bug
9 }

```

Code 5.1: Example of the relation between the bug types

In the `conditionalUpdate` method there are unsynchronized *combined atomic accesses*, and because of this, the operation in the `addAndGet` method is marked as a *simple atomic access* bug.

5.2 Results of detection the VNA04-J rule

The test results for the *VNA04-J* rule on large, opens-source projects are given below in the Table 5.2. In this case, there were no hits on the projects because in the final stage of the development, I decided to exclude the main thread analysis because the checker would have been so noisy that it is unusable (see the details in Chapter 4.3).

It is very rare that one class has 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. 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.

Project	Lines of Java Code	Count of combined access bugs	Count of simple access bugs
Bt	80714	0	0
MATSim-Libs	154975	0	0
Jenkins	311151	0	0
OpenGrok	146324	0	0
Ttorrent	21420	0	0

Table 5.2: VNA04-J Measurements on large, open source projects

5.3 Comparative analysis of existing static analyzer tools

With the help of the wide range of test cases which I developed for the SpotBugs testing framework, I perform a comparative analysis with other existing static analyzer. I run the following tools on all my test cases and I expect hits on the positive ones.

- *PMD* is a static code analysis tool that identifies bugs, unused code, and complexity across several programming languages, including Java. It allows for customizable rules to enhance code quality; [40]
- *SonarQube* is an automated code review tool that detects bugs, code smells, and vulnerabilities in over 20 programming languages, offering actionable insights through its interface; [38]
- *The Checker Framework* extends Java's type system to include pluggable type-checking, improving error detection and prevention at compile-time with type annotations; [41]
- *Google's Error Prone* is designed for Java, catching common programming mistakes at compile-time by focusing on patterns likely to lead to bugs. [42]

On my test cases there should be 22 hits on the *VNA03-J* cases and 10 hits on the *VNA04-J*, but it seems like these tools do not detect these multithreading atomicity rules, as can be seen in Table 5.3.

Name of the tool	Count of VNA03-J hits	Count of VNA04-J hits
PMD	0	0
SonarQube	0	0
The Checker Framework	0	0
Google's Error Prone	0	0

Table 5.3: Result of static analyzer tools hits on *VNA03-J* and *VNA04-J* test cases

SonarQube and PMD report atomicity-related hits, but this rule (AvoidUsingVolatile) [53] only focuses on the usage of the `volatile` keyword and suggests replacing it with a Java-built-in atomic type. As described in Section 2.3.1, the `volatile` keyword on non-primitive type is not enough to make that kind of fields usage atomic everywhere. This rule could be a good extension to my work because these hits indicate that the developer should use Java-built-in atomic types, and if fields with these types are entered into the code, my detectors will validate the proper usages of them.

The *VNA03-J* SEI Cert Rule Wiki page mentions that the Coverity [39] and Parasoft Jtest [54] tools implement the rule, but unfortunately I was unable to obtain licenses. On Coverity's official site [55], I found that it covers both *VNA03-J* and *VNA04-J* rules, but I did not find any specific information. The same was true on the Parasoft Jtest site, where I did not find any related information to the SEI Cert rules. In conclusion, I have not been able to collect any clue that the tools may cover all the issues or just a part of them. However, SpotBugs is open source and free to use, so the detection of these bugs is distributed to a large community of developers.

5.4 Known limitations and possibilities for further development

This chapter highlights limitations in the current implementation of my SpotBugs detectors and suggests some possible opportunity to improve them.

5.4.1 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, the *VNA03-J* detector's hits will be false positives. The detector assumes that, if the developer used atomic-typed variables, then it is because it will be used in threads. This limitation could be solved by analyzing the relations of the classes (see Section 5.4.2), and exclude those public methods from the analysis which are not run in parallel way.

5.4.2 SpotBugs' class context analyzing limitation

SpotBugs operates within a single-class context, limiting its ability to analyze operations within a class only. 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 *VNA03-J* checker could exclude those public methods, which are not running in parallel threads, and the *VNA04-J* detector could include those public 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.

5.4.3 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.

```
1 private AtomicInteger count = new AtomicInteger(0);
2
3 // Thread 1:
4 public void incrementByOne() {
5     count.incrementAndGet(); // Safe atomic operation
6 }
7
8 // Thread 2:
9 public void incrementByTwo() {
10    count.addAndGet(2); // Another safe atomic operation
11 }
```

Code 5.2: 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 based on the results of the large open-source projects, I chose to leave this false positive chance in the detector because it is not noisy for the developers.

5.4.4 Challenges with Lambda Expressions

The introduction of Lambda Expressions [56] into Java 8 marked a significant milestone in the evolution of the language. This feature was one of the most anticipated addi-

tions 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 [57], 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 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 because of the different bytecode instruction, and it is only partially implemented.

Chapter 6

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.

Throughout my research, I analyzed practices in thread safety, not only in Java but also by reviewing methodologies in other programming languages, such as C, C++, and Perl. This cross-language analysis was instrumental in understanding the unique approaches and challenges each language presents in concurrent environments.

I delved into multiple coding guidelines and standards, such as the SEI Cert Coding Standards, which are pivotal in guiding developers toward safer coding practices. My research into these guidelines was not just theoretical; I applied part of these standards practically by implementing the corresponding checkers (which cover the *VNA03-J* and *VNA04-J* rules) in SpotBugs Static Analyzer Tool.

By integrating new detectors, my 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' capabilities, enabling it to catch subtle bugs that could otherwise lead to inconsistent states or even system failures in production environments.

The enhancements to SpotBugs that were developed as part of my research 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 early in the development cycle, developers can address issues before they manifest in deployed systems, reducing down-time and maintenance costs. These advantages also enable managers to reduce the use of project resources and financial expenditures.

Furthermore, my work underscores the value of community-driven open source projects in the evolution of software development tools. My 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.

Appendix A

Unsafe Builder pattern usage with multiple threads

```
1 public final class Message {
2
3     private String sender = "";
4     private String recipient = "";
5     private String body = "";
6
7     public Message setSender(String sender) {
8         this.sender = sender;
9         return this;
10    }
11
12    public Message setRecipient(String recipient) {
13        this.recipient = recipient;
14        return this;
15    }
16
17    public Message setBody(String body) {
18        this.body = body;
19        return this;
20    }
21 }
22
23 public class ExampleClient {
24
25     private final Message message = new Message();
26 }
```

```
27 public ExampleClient() {
28     Thread t1 = new Thread(() -> {
29         message.setSender("Alice")
30             .setRecipient("Bob")
31             .setBody("Hello Bob!"); // Not thread-safe
32         // Send the message
33     });
34     t1.start();
35
36     Thread t2 = new Thread(() -> {
37         message.setSender("Bob")
38             .setRecipient("Jane")
39             .setBody("Hello Jane!"); // Not thread-safe
40         // Send the message
41     });
42     t2.start();
43 }
44 }
```

Appendix B

Compliant solution of using Builder pattern with multiple threads

```
1 public final class Message {
2     private final String sender;
3     private final String recipient;
4     private final String body;
5
6     private Message(Builder builder) {
7         this.sender = builder.sender;
8         this.recipient = builder.recipient;
9         this.body = builder.body;
10    }
11
12    public static class Builder {
13        private String sender = "";
14        private String recipient = "";
15        private String body = "";
16
17        public static Builder newInstance() {
18            return new Builder();
19        }
20
21        private Builder() {}
22
23        public Builder setSender(String sender) {
24            this.sender = sender;
25            return this;
26        }
27    }
28 }
```

```
27
28     public Builder setRecipient(String recipient) {
29         this.recipient = recipient;
30         return this;
31     }
32
33     public Builder setBody(String body) {
34         this.body = body;
35         return this;
36     }
37
38     public Message build() {
39         return new Message(this);
40     }
41 }
42 }
43
44 public class ExampleClient {
45     private Message message;
46
47     public ExampleClient() {
48         Thread t1 = new Thread(() -> {
49             message = Message.Builder.newInstance()
50                 .setSender("Alice")
51                 .setRecipient("Bob")
52                 .setBody("Hello Bob!")
53                 .build();
54         });
55         t1.start();
56
57         Thread t2 = new Thread(() -> {
58             message = Message.Builder.newInstance()
59                 .setSender("Bob")
60                 .setRecipient("Jane")
61                 .setBody("Hello Jane!")
62                 .build();
63         });
64         t2.start();
65     }
66 }
```

Bibliography

- [1] Vincent Hendryanto Halim and Yudistira Dwi Wardhana Asnar. “Static Code Analyzer for Detecting Web Application Vulnerability Using Control Flow Graphs”. In: *2019 International Conference on Data and Software Engineering (ICoDSE)*. 2019, pp. 1–6. DOI: 10.1109/ICoDSE48700.2019.9092687.
- [2] Ishwari Aghav et al. “Automated static data flow analysis”. In: *2013 Fourth International Conference on Computing, Communications and Networking Technologies (ICCCNT)*. 2013, pp. 1–4. DOI: 10.1109/ICCCNT.2013.6726670.
- [3] Xiaowen Zhang, Ying Zhou, and Shin Hwei Tan. “Efficient Pattern-based Static Analysis Approach via Regular-Expression Rules”. In: *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2023, pp. 132–143. DOI: 10.1109/SANER56733.2023.00022.
- [4] *SEI CERT Coding Standards*. <https://wiki.sei.cmu.edu/confluence/display/seccode/SEI+CERT+Coding+Standards>. Accessed: 04 2024.
- [5] *Java virtual machine*. <https://www.artima.com/insidejvm/ed2/platindep.html>. Accessed: 04 2024.
- [6] *Official SpotBugs website*. <https://spotbugs.github.io>. Accessed: 04 2024.
- [7] *Apache Commons BCEL*. <https://commons.apache.org/proper/commons-bcel/>. Accessed: 04 2024.
- [8] *VNA03-J. Do not assume that a group of calls to independently atomic methods is atomic*. <https://wiki.sei.cmu.edu/confluence/display/java/VNA03-J.+Do+not+assume+that+a+group+of+calls+to+independently+atomic+methods+is+atomic>. Accessed: 04 2024.
- [9] *VNA04-J. Ensure that calls to chained methods are atomic*. <https://wiki.sei.cmu.edu/confluence/display/java/VNA04-J.+Ensure+that+calls+to+chained+methods+are+atomic>. Accessed: 04 2024.

- [10] *Linux manual page of pthreads*. <https://www.man7.org/linux/man-pages/man7/pthreads.7.html>. Accessed: 04 2024.
- [11] *The Single UNIX Specification, Version 2, lock and unlock a mutex*. https://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread_mutex_lock.html. Accessed: 04 2024.
- [12] *The Single UNIX Specification, Version 2, wait on a condition*. https://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread_cond_wait.html. Accessed: 04 2024.
- [13] *The Single UNIX Specification, Version 2, lock a read-write lock object for reading*. https://pubs.opengroup.org/onlinepubs/7908799/xsh/pthread_rwlock_rdlock.html. Accessed: 04 2024.
- [14] *The GNU C Library Reference Manual, for version 2.38 - Semaphores*. https://www.gnu.org/software/libc/manual/html_node/Semaphores.html. Accessed: 04 2024.
- [15] *Concurrency support library (since C++11)*. <https://en.cppreference.com/w/cpp/thread>. Accessed: 04 2024.
- [16] *C++ - std::thread*. <https://cplusplus.com/reference/thread/thread/>. Accessed: 05 2024.
- [17] *C++ - std::mutex*. <https://cplusplus.com/reference/mutex/mutex/>. Accessed: 05 2024.
- [18] *C++ - std::condition_variable*. https://cplusplus.com/reference/condition_variable/condition_variable/. Accessed: 05 2024.
- [19] *cplusplus.com - std::future*. <https://cplusplus.com/reference/future/future/>. Accessed: 04 2024.
- [20] *cplusplus.com - std::promise*. <https://cplusplus.com/reference/future/promise/>. Accessed: 04 2024.
- [21] *Standard C++ - Discussion about std::thread and RAII*. <https://isocpp.org/files/papers/p0206r0.html>. Accessed: 05 2024.
- [22] *CppCoreGuidelines - E.6: Use RAII to prevent leaks*. <https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md>. Accessed: 04 2024.

- [23] *C++ reference - RAII*. <https://en.cppreference.com/w/cpp/language/raii>. Accessed: 04 2024.
- [24] *C++ - std::atomic*. <https://cplusplus.com/reference/atomic/atomic/>. Accessed: 04 2024.
- [25] *The Java Memory Model*. <https://www.cs.umd.edu/~pugh/java/memoryModel/>. Accessed: 04 2024.
- [26] *The Java Language Specification – Java SE 21 Edition*. <https://docs.oracle.com/javase/specs/jls/se21/jls21.pdf>. Accessed: 04 2024.
- [27] *Synchronized Methods – Oracle Java Documentation*. <https://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html>. Accessed: 04 2024.
- [28] *Package java.util.concurrent*. <https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/package-summary.html>. Accessed: 04 2024.
- [29] *Interpreter threads – Perl documentation*. <https://perldoc.perl.org/perlthrtut>. Accessed: 04 2024.
- [30] *threads::shared – Perl documentation*. <https://perldoc.perl.org/threads::shared>. Accessed: 04 2024.
- [31] *Thread::Semaphore – Perl documentation*. <https://perldoc.perl.org/Thread::Semaphore>. Accessed: 04 2024.
- [32] Hlib A. Nekrasov. “Coding Standards and Quality Checks: Key Factors for Successful Software Development”. In: *2023 International Conference on Quality Management, Transport and Information Security, Information Technologies (IT&QM&IS)*. 2023, pp. 19–22. DOI: 10.1109/ITQMTIS58985.2023.10346356.
- [33] *MISRA Coding Guidelines*. <https://misra.org.uk/misra-c/>. Accessed: 04 2024.
- [34] *OWASP Developer Guide*. <https://owasp.org/www-project-developer-guide/release/>. Accessed: 04 2024.
- [35] *Google Style Guides*. <https://google.github.io/styleguide/>. Accessed: 04 2024.
- [36] *The OWASP Top Ten*. <https://www.owasptopten.org>. Accessed: 05 2024.

- [37] *SEI CERT Oracle Coding Standard for Java*. <https://wiki.sei.cmu.edu/confluence/display/java/SEI+CERT+Oracle+Coding+Standard+for+Java>. Accessed: 05 2024.
- [38] *Official page of SonarSource*. <https://www.sonarsource.com/products/sonarqube/>. Accessed: 05 2024.
- [39] *Official page of Coverity Scan*. <https://scan.coverity.com>. Accessed: 05 2024.
- [40] *Official page of the PMD cross-language static code analyzer*. <https://pmd.github.io>. Accessed: 05 2024.
- [41] *Official page of The Checker Framework*. <https://checkerframework.org>. Accessed: 05 2024.
- [42] *Official page of Error Prone*. <https://errorprone.info>. Accessed: 05 2024.
- [43] *Refactoring Guru – Builder*. <https://refactoring.guru/design-patterns/builder>. Accessed: 04 2024.
- [44] *Java SE Documentation – Class StringBuilder*. <https://docs.oracle.com/javase/8/docs/api/java/lang/StringBuilder.html>. Accessed: 04 2024.
- [45] *Java SE Documentation – Class Thread*. <https://docs.oracle.com/javase/8/docs/api/java/lang/Thread.html>. Accessed: 04 2024.
- [46] *Java SE Documentation – Class Collections*. <https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html>. Accessed: 04 2024.
- [47] *Java SE Documentation – Class Runtime*. <https://docs.oracle.com/javase/8/docs/api/java/lang/Runtime.html>. Accessed: 04 2024.
- [48] *Bit Torrent*. <https://github.com/muddlebee/java-bt>. Accessed: 04 2024.
- [49] *MATSim*. <https://github.com/matsim-org/matsim-libs>. Accessed: 04 2024.
- [50] *Jenkins*. <https://github.com/jenkinsci/jenkins>. Accessed: 04 2024.
- [51] *OpenGrok*. <https://github.com/oracle/opengrok>. Accessed: 04 2024.
- [52] *Ttorrent*. <https://github.com/ttorrent/ttorrent>. Accessed: 04 2024.
- [53] *PMD Source Code Analyzer Project Documentation – AvoidUsingVolatile*. https://docs.pmd-code.org/latest/pmd_rules_java_multithreading.html#avoidusingvolatile. Accessed: 05 2024.

- [54] *Official page of Parasoft – Static Code Analysis Solutions & Tools for Compliance.* <https://www.parasoft.com/solutions/static-code-analysis/>. Accessed: 05 2024.
- [55] *Official page of Coverity Scan – Coverity Support for SEI CERT C, C++, and Java Coding Standards.* <https://www.synopsys.com/software-integrity/static-analysis-tools-sast/sei-cert.html>. Accessed: 05 2024.
- [56] *Java SE Documentation – Lambda Expressions.* <https://docs.oracle.com/javase/tutorial/java/java00/lambdaexpressions.html>. Accessed: 05 2024.
- [57] *Java SE Documentation – FunctionalInterface.* <https://docs.oracle.com/javase/8/docs/api/java/lang/FunctionalInterface.html>. Accessed: 05 2024.

List of Codes

2.1	Use Mutex to guard a section of the code	6
2.2	An example of using a condition variable in conjunction with mutex	6
2.3	Example of using Readers-Writer Locks	7
2.4	Example of using Semaphores	7
2.5	Use RAII C++ programming technique [23]	8
2.6	Example of a <code>std::atomic</code> variable	8
2.7	Atomic access of a volatile variable	9
2.8	Using a resource thread-safe with the <code>synchronized</code> keyword	10
2.9	Atomic usage of an <code>AtomicInteger</code> field	10
2.10	Sharing a variable between threads in Perl	11
2.11	Usage of semaphores in Perl	12
3.1	An example code of a field assignation	18
4.1	Combine Java atomic typed variables unsafe	21
4.2	Unsafe multiple operations on atomic variable	22
4.3	Unsafe access to thread-safe collection elements	22
4.4	Example usage of <code>StringBuilder</code> class in Java	23
4.5	Unsafe operation in multiple threads to a shared resource	23
4.6	Create a synchronized list	28
4.7	Synchronized private method on the call side	29
5.1	Example of the relation between the bug types	32
5.2	Example of safe usage of parallel modifications	35