YaleNUSCollege

Towards User-Friendly Linearizability Checking

Alaukik Nath Pant

Capstone Final Report for BSc (Honours) in
Mathematical, Computational and Statistical Sciences
Supervised by: Associate Professor Ilya Sergey
AY2020-2021

Yale-NUS College Capstone Project

DECLARATION & CONSENT

- I declare that the product of this Project, the Thesis, is the end result of my own work and that due acknowledgement has been given in the bibliography and references to ALL sources be they printed, electronic, or personal, in accordance with the academic regulations of Yale-NUS College.
- I acknowledge that the Thesis is subject to the policies relating to Yale-NUS College Intellectual Property (Yale-NUS HR 039).

ACCESS LEVEL

- I agree, in consultation with my supervisor(s), that the Thesis be given the access level specified below: [check one only]

✓ Unrestricted access
Make the Thesis immediately available for worldwide access.

Access restricted to Yale-NUS College for a limited period
Make the Thesis immediately available for Yale-NUS College access only from _____ (mm/yyyy) to _____ (mm/yyyy), up to a maximum of 2 years for the following reason(s): (please specify; attach a separate sheet if necessary):
_____.

After this period, the Thesis will be made available for worldwide access.

Other restrictions: (please specify if any part of your thesis should be restricted)
_____
_____

Alaukik Nath Pant - Cendana College
Name & Residential College of Student

Alaukik Nath Pant
Signature of Student

2nd April 2021
Date

Ilya Sergey
Name & Signature of Supervisor

2nd April 2021
Date

Acknowledgements

This capstone was possible because of the support of faculty, friends, family and the open source community.

In particular, I would like to thank Dr. Ilya Sergey not only for his advice on every aspect of this capstone, but also his inspiring foundational Computer Science courses at Yale-NUS College.

I would like to thank Dr. Michael Emmi for building Violat, the tool that I build on top of, and for generously giving his time to help me understand how Violat runs and how to debug it.

I would also like to thank my friends - Max, Michael, Gabriel, Ryan, Leyli, Adair and Karolina - for making my residential experience at 12-505, Cendana College a memorable one and for their constant emotional support.

Lastly, I would like to thank my family for inspiring and supporting me to be able to delve into a year-long, self-directed project.

YALE-NUS COLLEGE

Abstract

Mathematical, Computational and Statistical Sciences

B.Sc (Hons)

Towards User-Friendly Linearizability Checking

by Alaukik Nath PANT

The technology landscape is quickly moving towards multi-processor and multi-core computer systems. Hence, multi-threaded software design is becoming increasingly popular [4]. Multi-threaded software design often requires implementing Abstract Data Types(ADTs) that represent concurrent objects such as hashmaps, queues, and skiplists [4]. However, such ADTs are prone to bugs as they may be accessed by multiple processes and threads at the same time. Programmers can, therefore, reason about the correctness of their concurrent data types using linearizability as a correctness property.

There are several tools developed in academia to detect linearizability violations, but such tools are often hard to incorporate in every day use and industrial code bases. In this project, we integrate and extend one such tool, Violat, to the IntelliJ IDEA Integrated Development Environment (IDE) as a plugin. We achieve a plugin that fully automates testing linearizability violations in Java code bases using IntelliJ IDE.

Contents

List of Tables

List of Figures

Chapter 1

Background and Motivation

Concurrency bugs in software development are costly and hard to detect. However, concurrency remains important in the near future.

Concurrency's importance stems from the the current reality of Moore's law, i.e. the prediction that the transistor count per chip, which is a measure of a computer's processing performance, will double in approximately every two years [14]. While Moore's Law has not been broken, we cannot continue to add more transistors in the same space every year without overheating [8]. As a result, the field of shared-memory multiprocessors, i.e. multicores, has become increasingly important in Computer Science. Multicore processors enable the concurrent programming of software, which refers to writing code that splits into multiple tasks that can be executed concurrently on the same chip or different chips [7].

Multi-core architectures serve as an an "inflection" point in the software development industry because of the difficulties in multi-core programming [19]. In particular, tasks can be delayed or stopped in today's computer systems without warnings due to interruptions, cache misses and a variety of other reasons [8]. As a result, multiple sequences of operations can run in overlapping periods of time, and hence, it is challenging

```
public class CounterWrong {
    private int c;
    public CounterWrong() {c = 0};
    public int incrementAndGet() return ++c;
}
```

Fig. 1.1. Implementation of `CounterWrong` class in Java [2].

Thread A	Thread B
[A] read c (c=0)	
	[B] read c (c=0)
[A] incrementAndGet() (c=1)	
	[B] incrementAndGet() (c=1)

Table 1.1. Concurrent invocations of the `incrementAndGet()` method.

to figure out how to write correct concurrent code.

Consider the `CounterWrong` class ADT implementation in Figure 1.1. Suppose threads *A* and *B* are both trying to run the `incrementAndGet()` method. It seems correct that `c` should be 2 at the end of both operations. However, it is possible that both *A* and *B* read `c` as 0 before either of them write to it. As a result, as shown in the Table 1.1, we might get `c=1` instead of `c=2` at the end of both the operations.

Programming high-performance software requires more complicated and optimized implementations of common ADTs than the `CounterWrong` shown in Figure 1.1, which can lead to bugs that are even more difficult to detect.

1.1 Linearizability

Given the difficulty of writing concurrent code, specifying and verifying a given program helps ensure program correctness. Linearizability is the

key correctness property of concurrent objects. We now present definitions that will help us understand linearizability.

1. Action

   An action is an invocation or return of a particular method for a given thread [16].

2. Operation

   An operation is a pair of call action and return action where the call action and the return action are matching, i.e. are applied on the same object and thread [16]. For example, the invocation call to the `incrementAndGet()` method in Figure 1.1 and its return are two separate examples of an action and the pair is considered an operation.

3. History and Sub-history

   A history is a sequence of actions. A sub-history of a history, H, is a sub-sequence of the actions of H [16].

4. Sequential History and Well Formed History

   A history, H, is sequential if its first action is an invocation and every invocation is immediately followed by a matching return and every return is followed by an invocation except possibly the last one [16]. A thread sub-history of H is a sequence of actions carried out by a given thread. A well-formed history's thread sub-histories are all sequential [16].

5. Sequential Specification

For a given object, a sequential specification is a set of sequential histories for that object. A sequential specification helps us identify whether the thread history of a single object is legal [7].

6. Linearizability

   A history associated with a data structure is linearizable if there is a schedule of operations, whose operations when executed return the same result as the history specified by the sequential specification of this data structure [16, 7].

In more intuitive terms, linearizability specifies what return values are allowed when multiple threads perform a schedule of operations given our knowledge of the expected return values of operations when performed sequentially. Consider the histories H_1 and H_2 in Figures 1.2-1.5 of a queue, a first in, first out (FIFO) data structure, generated using a linearizability visualizer [22].

A queue comes with two operations:

1. $Enqueqe(E)$ inserts an item in the queue.
2. $Dequeqe(D)$ removes and returns the oldest item to be enqueued into the queue.

In these histories, processes are represented as horizontal axes. Each action is represented as a small vertical tick and each operation is represented as a horizontal line.

Example 1: A linearizable history H_1.

We argue the history H_1 shown in Figure 1.2 is linearizable because we can sequence the events as shown in Figure 1.3:

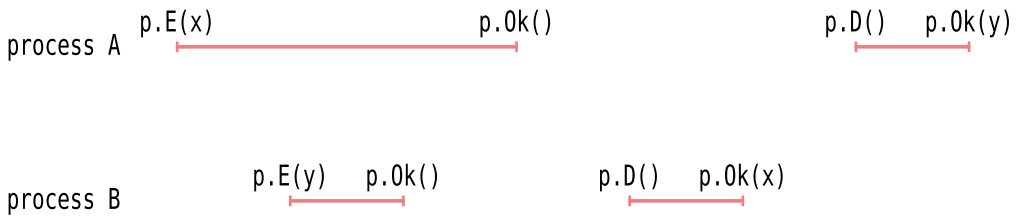

Fig. 1.2. History H_1.

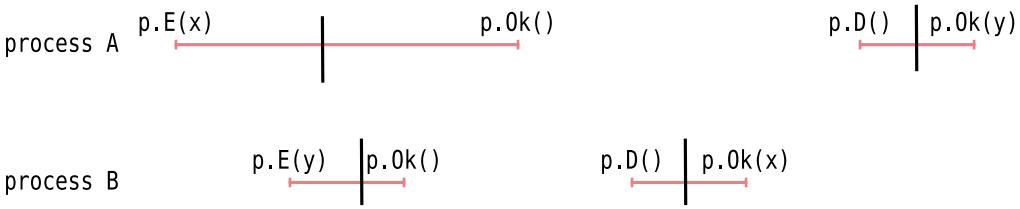

Fig. 1.3. History H_1 with linearization points.

The order of the events is as follows:

1. $p.E(x)$ - Enqueue x into the queue.
2. $p.E(y)$ - Enqueue y into the queue.
3. $p.D() = x$ - Dequeue and get x.
4. $p.D() = y$ - Dequeue and get y.

Note that the black vertical lines in Figure 1.3 represent the "linearization points", where the methods "take effect". Each operation has a linearization point at some point between its invocation and response [7]. We observe that enqueuing x and enqueuing y happen in overlapping periods of time, assuming time moves from left to right. We also observe that when we sequence our operations from left to right, i.e. create a sequential specification, we get x when we dequeue for the first time. To prove linearizability, we choose two points in the overlapping horizontal lines where each operation appears to “take effect” instantaneously such

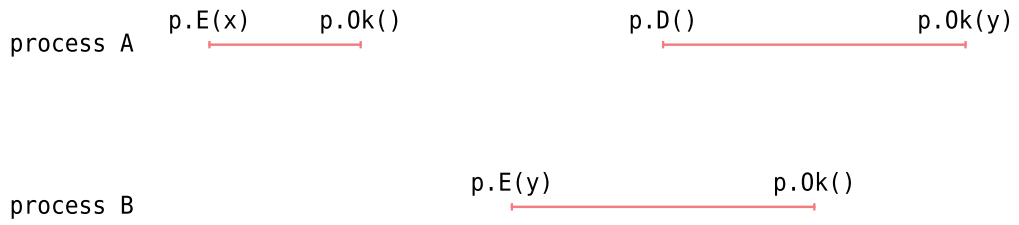

Fig. 1.4. History H_2.

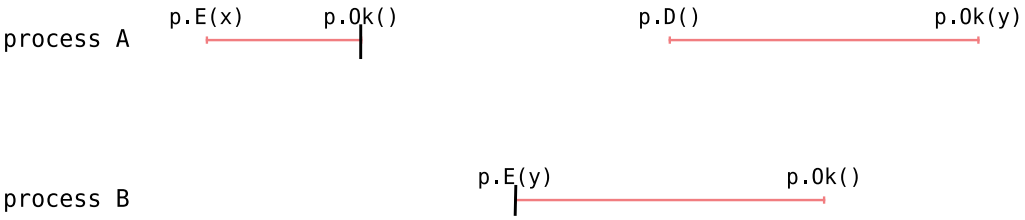

Fig. 1.5. History H_2 with linearization points.

that enqueuing x happens before enqueuing y. As a result, our schedule aligns with the sequential specification and the history H_1 is linearizable.

Since x and y are enqueued in overlapping periods of time, one can argue that we can pick linearization points such that y is enqueued before x. We argue that this history is still not considered non-linearizable because we just have to show one set of linearization points to prove the linearizability of an execution.

Example 2: A non-linearizable history H_2.

We argue the history H_2 shown in Figure 1.4 is non-linearizable. In this history, we notice that the first time we dequeue, we get a y. However, as shown in Fig 1.5 by the black vertical lines, even if we pick the latest possible linearization point in the first operation and the earliest possible point in the second operation, the operation of enqueuing y cannot occur before the one enqueuing x. Hence, history H_2 is non-linearizable.

Key Properties of Linearizability

1. Linearizability is a local property [7]. A system is linearizable if and only if each object belonging to this system is linearizable.

2. Linearizability is more of a property of an execution than that of an object [7]. An object is linearizable if all its possible executions are linerizable.

3. Linearizability is a non-blocking property. A pending invocation does not have to wait for another pending invocation to return because each invocation except the last one is immediately followed by a return in a linearizable history [8].

1.2 Contributions

Writing concurrent code often requires implementing optimized implementations of concurrent ADTs that may or may not be linearizable. While there are various tools to check the linearizability of concurrent ADTs produced in academia [1, 4, 16], these tools do no have much adoption in industry [13]. We present ViolatIntegration, a plug-in for IntelliJ IDEA, which we developed in order to *fully automate* the process of linearizability checking for user-defined ADTs.

In summary, we present the following contributions.

1. An investigation and experimentation with several state of the art tools for linearizability testing;

2. a comparison of Violat with other tools;

3. an extension of Violat to a larger class of programs;

4. an integration of Violat to the popular IDE for Java, IntelliJ IDEA, as a plugin called ViolatIntegration;

5. a tutorial on using ViolatIntegration to validate concurrent programs.

1.3 Paper Outline

In the following chapters, we describe the principles and limitations of various academic tools and our rationale for picking Violat (Chapter 2); present ViolatIntegration through an example as a plugin for IntelliJ IDEA that fully automates linearizability checking (Chapter 3); describe some of the interesting details that went into extending Violat and integrating it into a plugin (Chapter 4); and end with a comparison of ViolatIntegration with another industrial tool and a discussion of future work (Chapter 5).

Chapter 2

Tools for Linearizability Checking

In this chapter, we investigate two state-of-the-art tools for linearizability checking and choose a tool to improve.

2.1 On Linearizability Checking

The problem of verifying that an execution is linearizable by exhaustively searching for a schedule that is equivalent to the sequential specification of the same execution is NP-Complete. Hence, linearizability checking tools that check all possible schedules of a history perform poorly [16].

For this project, we reviewed several methods and tools developed in academia to detect linearizability violations in concurrent programs. In this report, we present two shortlisted state-of-the-art tools, Check-Lin [16] and Violat [4]. Ultimately, we build an IntelliJ IDEA plugin on top of Violat to detect linearizability violations for reasons discussed below.

2.2 Review of Tools

2.2.1 Check-Lin

Principle

Check-Lin is based on the empirical observation that linearizability of an execution is frequently seen in a sub-schedule [16]. For this reason, Check-Lin considers only a few of operations as opposed to exhaustively searching through all of them in each schedule. The number of operations in the aforementioned sub-schedule which witness linearizability is known as "linearizability depth". The algorithm starts to search for witnesses of linearizability at low linearizability depths before moving deeper [16]. To do so, Check-Lin generates a set of schedules that are guaranteed to find all linearizability witnesses at depth d called a "strong d-hitting family". The authors argue that if we check a strong d-hitting family with a linearizability depth $d \leq 5$, then that is enough to show linearizability for 99.9% of experimented linearizable traces [16]. Hence, we conclude that Check-Lin is efficient and practical for showing linearizability of an execution trace.

Shortcomings of Check-Lin

Check-Lin, written using the Scala programming language, can potentially be used to detect linearizability violations in several Java Virtual Machine(JVM)-based languages. However, as we discuss below, Check-Lin has some major drawbacks that make it an unfavourable candidate for an average user.

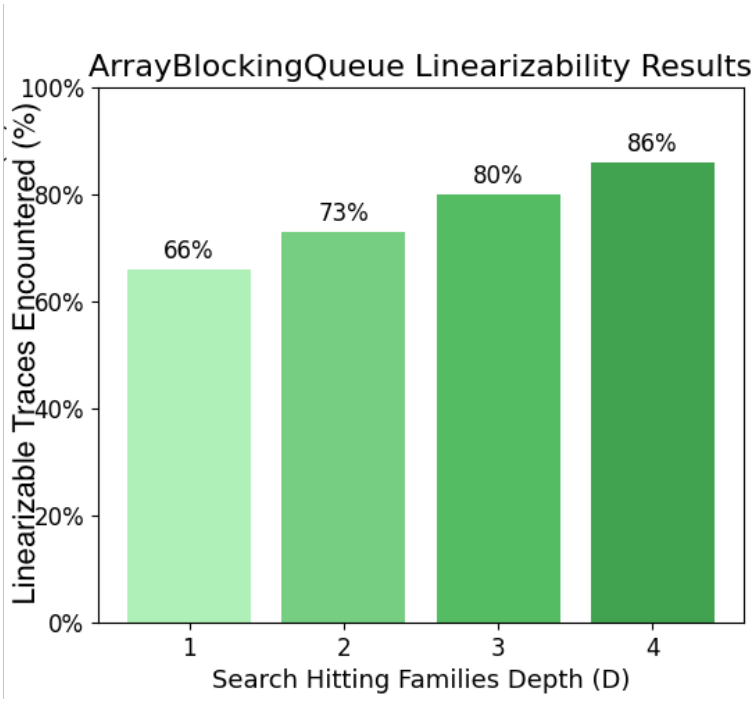

Fig. 2.1. Result for `ArrayBlockingQueue` using Check-Lin.

1. The main drawback of Check-Lin is that its result is hard to interpret. Figure 2.1 shows a sample result we generate using Check-Lin to check linearizability violations in an `ArrayBlockingQueue` from Java's `java.util.concurrent` package at different depths D. The result of Check-Lin shows what percentage of a family of a "strongly d-hitting family" of schedules witness linearizability at a depth D. For example, for a depth of 4 in Figure 2.1, this statistic is 86%. We believe that, for the average user, this result is not only difficult to interpret but also does not help them debug their code.

2. Check-Lin requires the user to provide a JSON specification file describing their ADT and there is insufficient instruction on how to write such specifications.

3. Check-Lin is not well tested on user-defined ADTs. In fact, there are

no examples of the authors using the tool on user-defined ADTs, outside of those found on Java's `java.util.concurrent` package.

2.2.2 Our Choice: Violat

Principle

Violat is a tool that generates tests of observational refinement for concurrent objects and uses those tests to discover violations to linearizability. An implementation of a data structure O is said to be an observational refinement of another implementation O' if every behaviour of a program using O can be observed using O' [6]. Tests of observational refinement coincide with tests of linearizability [6].

To generate tests of observational refinement, Violat takes a three step approach, which we enumerate below.

1. Violat generates a schema that has descriptions of parallel sequences of invocations of the methods associated with a given object where each invocation is separated by "`||`". For example, for the `CounterWrong` class in Figure 1.1, a schema would look like the following:

   ```
   incrementAndGet(); incrementAndGet() || incrementAndGet().
   ```

2. Violat sequentially executes the methods of the given ADT implementation and labels each such schema with expected outcomes calculated from the result of these executions.[4].

3. Violat creates a self contained Java class from each schema [4] and tests the classes created with two back-end analysis engines: i) Java Concurrency Stress testing tool(JCStress) [18] and Java Pathfinder[21].

The aforementioned back-end analysis engines give the output of the execution traces that lead to linearizability violations, which helps the user identify the source of their mistakes and amend them. For this reason, Violat's output is more user-friendly than Check-Lin..

Shortcomings of Violat

Violat, however, is not entirely suitable for a developer to seamlessly test their ADT implementation for reasons discussed below.

1. For the first two steps of test generation, Violat requires a specification of the given concurrent object describing the methods and constructor in JSON format. While Violat's paper mentions that this specification can be generated from an object's byte-code [4], it does not provide a command to do this.

2. Violat, like Check-Lin, is not well tested. There are no examples of Violat being used on ADTs outside of those already packaged with Java. Our own experiments reveal that Violat does not work correctly for simple user-defined Java classes due to unexpected interactions between Violat and JCStress [17].

3. Violat requires the Java class being tested in the system classpath or in a user-provided Java archive (JAR). Ideally, for wide spread use, we would just want to click on a class and get the required output.

While Violat may not be user-friendly, its output helps the user identify the execution trace that led to linearizability violations and amend their ADT implementation. Hence, we decide that Violat better serves the average user than Check-Lin, whose output is a score that is hard

to interpret. If we can create a tool on top of Violat that automates the process of checking linearizability violations, then that would reduce the barriers to test concurrent ADTs implemented in Java. Since IntelliJ IDEA IDE is a popular application to write Java code in with several features such as source code editing, build automation and a debugger, integrating Violat on top of it can be useful. While there are several IDEs such as Eclipse, Emacs and Kite to write Java code in, IntelliJ IDEA has robust developer support to write tools on top of it. In particular, IntelliJ has individual software components called plugins that are extensions that can add new functions to it. Hence, we implement a plugin, presented in the next chapter, for the IntelliJ IDEA IDE with the goal of completely automating linearizability checking.

Chapter 3

A tour of ViolatIntegration

In this chapter, using an example, we present ViolatIntegration as a plugin that fully automates the process of checking linearizability violations in ADTs written in Java. We then describe the results of using ViolatIntegration to test ADTs found in open source repositories.

3.1 Motivating Example

Consider the `QueueWrong` class shown in Figure 3.1, which we adapted from an open source implementation [2]. It represents a simple implementation of a first-in, first-out (FIFO) data structure, meaning the first integer to be put into `QueueWrong` using the `put(int)` method is also the first element to be removed by the `get()` method. Note that this queue uses a shared array called `items` with a capacity for 100 elements. Also note that the shared variable `indPut` stores the index where the `put(int)` operation will put an integer. Likewise, the shared variable `indGet` represents the index of the element we get from the `get()` operation. Both `indPut` and `indGet` are incremented each time we call their associated

```
public class QueueWrong {
    private int indGet;
    private int indPut;
    private int countElements;
    private int[] items;

    private int inc(int i) {
        return (++i == items.length ? 0 : i);
    }

    public QueueWrong() {
        items = new int[100];
        indPut = 0;
        indGet = 0;
        countElements = 0;
    }

    public void put(int x) throws Exception {
        if (countElements == items.length) {
            throw new Exception("Queue is full");
        }
        items[indPut] = x;
        indPut = inc(indPut);
        countElements++;
    }

    public int get() throws Exception {
        if (countElements == 0) {
            throw new Exception("Queue is empty");
        }
        int ret = items[indGet];
        indGet = inc(indGet);
        countElements--;
        return ret;
    }
}
```

Fig. 3.1. Implementation of `QueueWrong` in Java [2].

methods using the `inc()` method. Finally, the `countElements` shared variable keeps a count of the number of elements in the queue.

As the name suggests, the `QueueWrong` class has problems. We notice

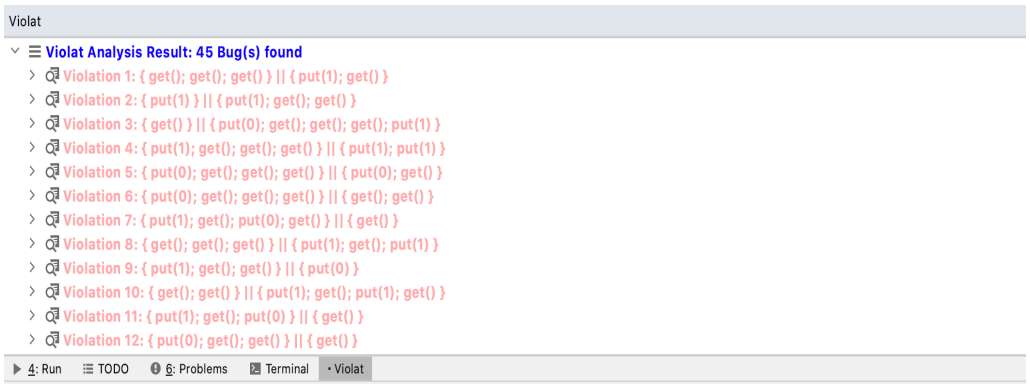

Fig. 3.2. Violations that Violat console shows.

that the shared memory locations held by the variables `indGet`, `indPut`, `countElements` and `items` can be concurrently accessed and modified. In other words, when one process is accessing the QueueWrong object, another process can change this object's contents through the aforementioned variables. Although operations can overlap on a shared object in a linearizable system, each operation appears to take effect instantaneously at its linearizaton point. Hence, the `QueueWrong` object might be problematic because its contents can be changed by another process while a given process is still accessing it.

We will check this data structure for linearizability violations using ViolatIntegration. A step-by-step tutorial describing the the configuration required to use ViolatIntegration to test an ADT like this is available at Appendix B. Once we set-up the required configuration for the `QueueWrong` class and run Violat, we get a total of 45 violations, as shown in the Violat Console at the bottom of Intellij IDEA (Figure 3.2). Each violation shows the combination of invocations that led to an error. Let us explore violation 16 shown in Figure 3.3 by clicking on it, and understand what it means.

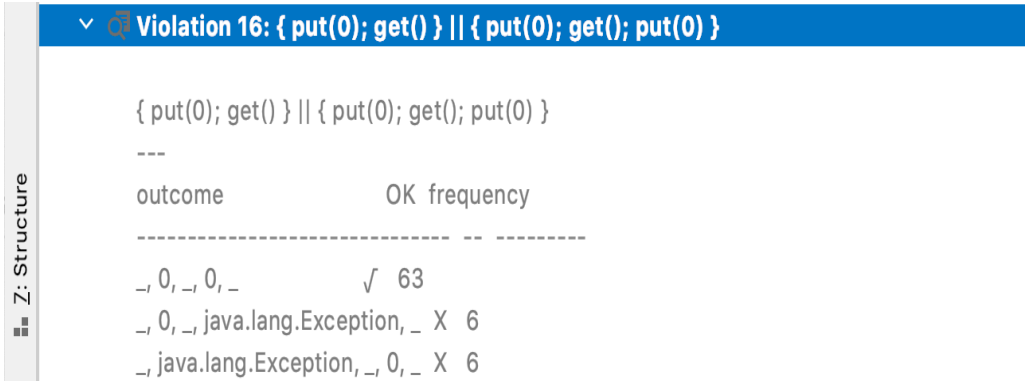

Fig. 3.3. A look into Violation 16.

The violation under consideration is represented by the following program schema:

```
put(0); get()|| put(0); get(); put(0).
```

Here, our schema has two parallel threads. The sequence of methods of the `QueueWrong` object invoked by each thread is separated by "`||`". The methods invoked by the first thread are written in red while those invoked by the second thread are written in blue.

We now shuffle all possible invocations of the methods associated with the aforementioned threads and record the return value of each shuffle [4]. Recall that the `put(0)` method invocation has a `void` return type, represented by a "_" string in the outcome list. Also recall that `get()` returns whatever was put into `QueueWrong` first. Both `get()` and `put()` return an `Exception` if the `QueueWrong` is empty or full respectively. Additionally, note that while we can shuffle operations invoked by different threads in any order, the order of the operations within each thread must be maintained. For example, `put(0)` from the first thread, whose method names are colored in red, always comes before `get()` from the

Number	Combination	Outcome
1.	[*put*(0), *get*(), *put*(0), *get*(), *put*(0)]	[_, 0, _, 0, _]
2.	[*put*(0), *put*(0), *get*(), *get*(), *put*(0)]	[_, _, 0, 0, _]
3.	[*put*(0), *get*(), *put*(0), *get*(), *put*(0)]	[_, 0, _, 0, _]
4.	[*put*(0), *get*(), *put*(0), *put*(0), *get*()]	[_, 0, _, _, 0]
5.	[*put*(0), *put*(0), *get*(), *get*(), *put*(0)]	[_, _, 0, 0, _]
6.	[*put*(0), *put*(0), *get*(), *get*(), *put*(0)]	[_, _, 0, 0, _]
7.	[*put*(0), *put*(0), *get*(), *put*(0), *get*()]	[_, _, 0, _, 0]
8.	[*put*(0), *put*(0), *get*(), *put*(0), *get*()]	[_, _, 0, _, 0]
9.	[*put*(0), *put*(0), *get*(), *get*(), *put*(0)]	[_, _, 0, 0, _]
10.	[*put*(0), *get*(), *put*(0), *put*(0), *get*()]	[_, 0, _, _, 0]

Table 3.1. Enumeration of sequential executions of `QueueWrong`.

same thread. There are 10 ways to shuffle the method invocations this way, as shown in the Table 3.1. We notice that there are only 4 unique outcomes, all highlighted in yellow in the Table 3.1. We can now annotate our schema with the 4 expected outcomes.

Next, we can generate tests that run the schema's threads and their associated methods in parallel. We can then record the result of each invocation and check if the recorded outcome is expected. This is done by one of the two selected analysis back-ends: the Java Concurrency Stress Testing Tool(JCStress) or Java Pathfinder(JPF).

3.1.1 Interpretation of result

Given the aforementioned context, we expect a developer to interpret the result in Figure 3.3 as follows:

To expose linearizability violations in the `QueueWrong` class, Violat explored 75 program paths. Out of the 75, 63 program paths gave "_, 0, _, 0, _" as their outcome, which aligned with the expected outcome of the given schema. However, 6 program paths had "_, 0, _, java.lang.Exception, _" as their outcome and

the remaining 6 had "_, 0, _, java.lang.Exception, _". Both of these outcomes do not align with the expected results found by enumerating sequential executions. This violation suggests that the implementation of `QueueWrong` *has executions that are not linearizable. This violation also suggests that there might be concurrency bugs in one or more of the methods mentioned in the schema, i.e* `put()` *and* `get()`*.*

3.1.2 How to fix your code based on the results

Since this is a relatively small program, we revisit the `get()` and `put()` methods in the `QueueWrong` Class in Figure 3.1. We know that when the tests invoke the schema's threads in parallel, the same memory locations held by variables such as `indGet`, `indPut`, `countElements` and `items` are concurrently accessed. An easy solution is to synchronize the call to the methods `get()` and `put()` on the current instance (obtain lock on the current instance) so that other processes cannot change the shared variables when a given process is accesing the `QueueWrong` object. In other words, we just add the synchronized modifier in front of `get()` and `put()` methods and Violat reports 0 violations after.

3.2 Experiments

Using a similar process to the one mentioned above, we also search for linearizability violations in ADT implementations written in Java in open source github repositories using our plugin. Prior to our plugin, Violat was only tested on classes found in `java.util.concurrent` package. This

Number	Class Name	Number of Violations
1.	AccountABA	76
2.	QueueWrong	45
3.	Account	0
4.	LazyList	1
5.	Sequence	85
6.	StampedAccount	56
7.	LinkedList	58
8.	MyHashMap	27
9.	NonBlocking	16
10.	QueueSynchronized	0
	Total	**365**

Table 3.2. Linearizability violations in open source repositories.

is the first time Violat is used to discover violations in user-defined ADTs and that too in a *fully automated* manner.

In Table 3.2, we list the number of violations we found for each of the 10 ADT implementations we picked . In the process of doing these experiments, we also found some bugs in the source code of Violat, which we have discussed in Section 4.4. The source to implementations of classes mentioned in Table 3.2, their associated repository, instructions on reproducing these results and the output that ViolatIntegration gives is available in Appendix A.

In summary, in this chapter, we present ViolatIntegration, using examples, as a user-friendly plugin for IntelliJ IDEA to check linearizability violations without going into the details of its implementation.

Chapter 4

Details of Implementation

This chapter gives a high-level description of the implementation of ViolatIntegration and describes some of the interesting details of extending Violat and integrating it into IntelliJ IDEA as a plugin.

4.1 Structure of ViolatIntegration

Most of the effort of this capstone was dedicated to building ViolatIntegration, which can be built from the source provided in Appendix A. ViolatIntegration is primarily written in Java with a few supporting features written in Kotlin. The plugin is designed using the IntelliJ Platform Plugin Template [10] as a base, which makes the the process of configuring the project scaffold, Continuous Integration(CI), testing and deployment easier. To build features on top of this template, we utilize the IntelliJ Platform SDK DevGuide [11]. We also take inspiration from the implementation and design of several open source projects, most notably the IntelliJ Platform SDK Code Samples [12] and Infer Integration [9], a plugin that integrates Facebook's Infer Static Analyzer tool [5], which detects data races at scale.

The most important sections in the implementation of the plugin is in the *src* directory of our code-base, which contains packages with implementations of the features of our plugin. We briefly describe each package and its contribution to the plugin implementation below.

1. `actions`: This package contains all actions. Each action invokes a functionality of the plugin.

2. `model`: This package contains classes that model Violat's Installation, checkers, testers, artifacts, build tools, specification, Violat's Version and Java classes representing ADTs. More information about each element being modelled can be found in Appendix B.

3. `pluginconfig`: This package contains classes that configure the plugin with information such as the path to a valid Violat installation at start-up time.

4. `resultparsers`: This package contains classes that parse the result of Violat.

5. `specgenerator`: This package contains classes that generate a specification (JSON file) from a class file.

6. `toolwindows`: This package contains classes that describe the UI of the various forms used in the plugin.

4.2 Developing a Specification Generator

In this section, we describe the aforementioned `specgenerator` package in detail. As mentioned in Section 2.2.2, Violat's final output depends

on the specification describing the implementation of the ADT including features such as the constructor and the method signatures as a JSON file. However, even for small ADTs, manually writing such a specification might end up being long and confusing. Hence, we extend Violat's functionality by writing a Specification Generator in this package.

The design of the `specgenerator` package entails using the Reflection Application Programming Interface(API) [15] and setting certain parameters that Violat expects as default values. In particular, once the user selects a class to test, the package loads the relevant class at run-time utilizing Java's Reflection API. Then, it extracts information such as the names of the methods associated with this concurrent object, the return types of the methods and the parameters of the methods. In the generated specification, the number of threads is set to the default value of 2 and the maximum number of invocations in each thread is set to be 3. While we can let the user input these values, the values to be inserted might not be intuitive. After all, theoretically, some concurrency bugs can only be found when using a large number of threads and invocations. Fortunately, it is reported that 96% of such bugs can be found with just 2 threads [4]. Hence, the aforementioned numbers of threads and invocations are sufficient and enable Violat to get a probabilistically correct output at a reasonable amount of time. In this way, we automate the generation of a specification describing an ADT.

4.3 Using PSI Files

To further automate linearizability checking, we use IntelliJ's feature called Program Structure Interface (PSI), which parses files and creates a syntactic and semantic code model [11]. A PSI file has information about the hierarchy of elements in a given programming language, which we use for the following purposes:

1. to get the name of the class along with the associated package when we select a user-defined ADT,

2. to find the path to the selected class using various packages that link PSI files with utilities for working with files,

3. to script command line commands and create an execution pipeline for Violat.

4.4 Bugs Discovered in Violat

Next, in order to create an automated execution pipeline that does not break, we deal with the shortcomings of Violat. Violat is an academic tool that is not well tested on data structures outside of those in the `java.util.concurrent` package. In this section, we enumerate the bugs found in Violat's source code and discuss their resolutions.

1. Violat sometimes gives an error alerting us that there are *"No valid entries in array weights"*. We raise this as an issue on Violat's source-code issue tracker [20], but a resolution is yet to be found. Keeping this issue in mind, we ensure that our plugin does not break when

we get this error by showing that no bugs have been found on the Violat console.

2. Violat gives a compilation error when the selected ADT implementation is not packaged and a run-time error when there is a number in the name of the ADT implementation [17]. The solution to the second problem is to change the regular expression associated with the expected name of the ADT implementation. We alert the author of Violat about this bug, but we ask the user to package their implementation and avoid numbers in the name of their ADT for now.

3. Our experiments also reveal that the *histories* checker fails non-deterministically for user-defined ADTs. Hence, we avoid adding this checker to the current implementation of the plugin.

Given these limitations of Violat, we develop features in the plugin that Violat currently supports.

4.5 Containerized Run-time Environments

We also provide a containerized run-time environment to run Violat. After all, end users of ViolatIntegration might struggle to reproduce the results that we get as it requires a Node.js run-time for JavaScript, Java-8, Gradle, Maven and Java Pathfinder as an executable in the user's path. In fact, some of the issues raised in Violat's github page are related to the required run-time environment. Since the aim of our plugin is to make the process of using Violat as seamless as possible, we provide a suitable

run-time environment using containers. A container solves the problem of getting software to run reliably in different computing environments. Docker [3] is a popular container that abstracts the problem of setting up a configuration for your project and also has support in the IntelliJ platform as a Docker Plugin [3]. There are two ways in which the user can setup Docker to run ViolatIntegration.

1. Docker Image - The user can build a docker image from the Dockerfile we provide and run the container from the existing image.

2. Dockerfile - The user can run a container from the Dockerfile itself, which can be done by building a image from the Dockerfile and using the associated container of this image.

Alternatively, if the user wishes to build ViolatIntegration from the source, we provide a Dockerfile that helps the user run an instance of the IDE with Violat installed. More instructions on how to do this is found on the **develop** branch of the associated github repository.

In summary, in this chapter, we discuss our use of APIs, PSI files, containers and debugging techniques to deliver a fully automated linearizability checking plugin.

Chapter 5

Discussion

In this chapter, we compare ViolatIntegration with the only other similar industrial tool available for JVM-based languages. We then conclude with a discussion of future work and the contribution of ViolatIntegration in making linearizability checking more accessible.

5.1 Comparison with Lin-Check

The only other user friendly tool to expose linearisability violations on JVM-based languages that we found is Lin-Check [2], different from Check-Lin mentioned in Section 2.2.1. In this section, we compare Lin-Check with ViolatIntegration on different grounds.

5.1.1 Process of testing

The process of testing an ADT is simpler in ViolatIntegration. To use Lin-Check, the user has to download Lin-Check's artifact and add it as a dependency and manually write tests for their ADT implementations. Lin-Check requires detailed information about the execution environment such as the number of threads to use and different scenarios to

```
@StressCTes(name = "key", gen = IntGen.class, conf = "1:5")
@StressCTest
public class HashMapLinearizabilityTest {
    private HashMap<Integer, Integer> map = new HashMap<>();;

    @Operation
    public Integer put(@Param(name = "key") int key, int value) {
        return map.put(key, value);
    }
    ...

    @Test
    public void test() {
        LinChecker.check(HashMapLinearizabilityTest.class);
    }
    ...
}
```

Fig. 5.1. Example input program for Lin-Check [2].

execute [2]. All of this information can be fed to Lin-Check with annotations such as "@StressCTest" and "@Operation". For example, to test an implementation of a *HashMap* using Lin-Check, part of the input that the user has to give is shown in Figure 5.1. On the other hand, ViolatIntegration does not require any written inputs from the user while providing meaningful information about the execution traces that led to linearizability violations. All that the user has to do is to download the plugin from the JetBrains marketplace, select the class that they want to test and click a few buttons on their IDE. Hence, we argue that the process of testing an ADT for linearizability violations is less complicated using ViolatIntegration. However, we acknowledge that Lin-Check gives a good framework to test only certain methods and is more customizable at the cost of complexity.

5.1.2 Performance

Since Lin-Check was specifically designed as an industrial tool, it reveals violations much faster and it always looks for the first possible violation. In ViolatIntegration, we do give the option to only check 1 program path, but this does not necessarily give a violation in nuanced data structures where bugs are harder to detect. However, Lin-Check does not have the feature to show multiple violations like ViolatIntegration. Hence, in comparison to Violat, Lin-Check trades speed for accuracy.

5.1.3 Results

Although presented differently, both Lin-Check and Violat give us meaningful results that inform us which methods, when run concurrently, lead to linearizability violations. After-all, both Violat and Lin-Check share a common back-end analysis engine - JCStress. Hence, we argue that both Lin-Check and Violat are comparable in terms of the presentation of results.

5.2 Future Work

The focus of this project was to establish an infrastructure on top of Violat to automate the process of checking linearizability by building a working plugin in the IntelliJ IDE. We achieved this goal and successfully passed all of JetBrain's requirements to publish the initial version of our plugin manually in the marketplace. Given the successful deployment, releasing future versions of the plugin will be simpler. Hence, further work can focus on the following areas:

1. Fix the bugs associated with Violat mentioned in Section 4.4.
2. Once the *histories* checker has been fixed, add visualisation effects for checking linearizability within IntelliJ.
3. Build a user interface for the user to conveniently change the generated specification so that the decision on what methods and the number of threads/invocations to use are easy to specify.
4. Make the user interface of ViolatIntegration more responsive.

We may also add support for Violat for ADTs written in other JVM-based languages, i.e. Scala and Kotlin, using IntelliJ PSI. Additionally, we may consider adding support for Lin-Check in the same plugin to give the user multiple options to check their ADT implementation for linearizability violations.

5.3 Conclusion

In this work, we introduce the difficulties of testing concurrent programs and present a plugin that fully automates checking the linearizability correctness condition. We explore two state-of-the-art academic linearizability checking tools, Lin-Check and Violat. We argue that Violat, though not entirely user-friendly, is more useful as it gives the specific execution trace that led to a violation. We then describe our work in extending Violat by adding a specification generator on top of it. Finally, we outline our discoveries and experience engineering ViolatIntegration using IntelliJ's plugin development framework.

The experiments in Section 3.2, where we reveal linearizability violations in ADT implementations from multiple open source repositories, shows that mainstream developers can now test their Java classes easily and in an automated fashion using ViolatIntegration. As a result, in addition to the more customizable Lin-Check industrial tool, users now have the option to check linearizability violations in their ADT implementations in a fully automated manner using ViolatIntegration.

We hope for a wider adoption of ViolatIntegation in industrial and personal code bases. Our expectation is that using ViolatIntegation will potentially make the arduous process of writing correct concurrent code easier. As of 4^{th} April 2021, we see 40 downloads in JetBrains marketplace and hope for more downloads. We have even published ViolatIntegration as an open source repository on github. We welcome developers to extend its functionality or edit it in other meaningful ways that will make linearizability checking more user-friendly.

Bibliography

[1] Sebastian Burckhardt, Chris Dern, Madanlal Musuvathi, and Roy Tan. "Line-Up: A Complete and Automatic Linearizability Checker". In: *ACM SIGPLAN Notices*. Vol. 45. 2010, pp. 330–340 (cit. on p. 7).

[2] Devexperts. *Lin-Check*. https://github.com/devexperts/lin-check. Last Accessed: 2021-03-24 (cit. on pp. 2, 15, 16, 28, 29).

[3] Docker. *Empowering App Development for Developers*. https://www.docker.com/. Last Accessed: 2021-03-16 (cit. on p. 27).

[4] Michael Emmi and Constantin Enea. "Violat: Generating Tests of Observational Refinement for Concurrent Objects". In: *Computer Aided Verification(CAV)*. 2019, pp. 534–546 (cit. on pp. iii, 7, 9, 12, 13, 18, 24, 41).

[5] Facebook. *Infer Static Analyzer*. https://fbinfer.com/. Last Accessed: 2021-03-13 (cit. on p. 22).

[6] Ivana Filipović, Peter O'Hearn, Noam Rinetzky, and Hongseok Yang. "Abstraction for concurrent objects". In: *Theoretical Computer Science* 411.51-52 (2010), 4379–4398 (cit. on p. 12).

[7] Maurice Herlihy. *Art of Multiprocessor Programming*. Elsevier Science amp; Technology, 2008, pp. 45–48 (cit. on pp. 1, 4, 5, 7).

[8] Maurice Herlihy and Jeannette Wing. "Linearizability: A Correctness Condition for Concurrent Objects". In: *ACM Transactions on Programming Languages and Systems* 12 (1990), pp. 463– (cit. on pp. 1, 7).

[9] Friedrich Hudinjan. *Infer Integration - Plugins: JetBrains*. https://plugins.jetbrains.com/plugin/12847-infer-integration. Last Accessed: 2021-03-17 (cit. on p. 22).

[10] JetBrains. *Intellij Platform-plugin-template*. https://github.com/JetBrains/intellij-platform-plugin-template#plugin-template-structure. Last Accessed: 2021-03-14 (cit. on p. 22).

[11] JetBrains. *IntelliJ Platform SDK - Help*. https://plugins.jetbrains.com/docs/intellij/welcome.html. Last Accessed: 2021-03-21 (cit. on pp. 22, 25).

[12] JetBrains. *IntelliJ Platform SDK Code Samples*. https://github.com/JetBrains/intellij-sdk-code-samples. Last Accessed: 2021-03-21 (cit. on p. 22).

[13] Nikita Koval, Maria Sokolova, Alexander Fedorov, Dan Alistarh, and Dmitry Tsitelov. "Testing concurrency on the JVM with lincheck". In: *PPoPP '20: 25th ACM SIGPLAN, California, USA, February 22-26, 2020*. ACM, 2020, pp. 423–424 (cit. on p. 7).

[14] Ethan Mollick. "Establishing Moore's Law". In: *Annals of the History of Computing, IEEE* 28 (Aug. 2006), pp. 62 –75. DOI: 10.1109/MAHC.2006.45 (cit. on p. 1).

[15] Oracle. *The Reflection API*. https://docs.oracle.com/javase/tutorial/reflect/index.html. Last Accessed: 2021-03-29 (cit. on p. 24).

[16] Burcu Kulahcioglu Ozkan, Rupak Majumdar, and Filip Niksic. "Checking linearizability using hitting families". In: *Proceedings of the 24th ACM SIGPLAN*. ACM, 2019, pp. 366–377 (cit. on pp. 3, 4, 7, 9, 10).

[17] Alaukik Pant. *Violat Tests*. https://github.com/alaukiknpant/violatTests. Last Accessed: 2021-03-10. 2021 (cit. on pp. 13, 26).

[18] Aleksey Shipilev. *Code Tools: jcstress*. https://wiki.openjdk.java.net/pages/viewpage.action?pageId=42598465. Last Accessed: 2021-03-24 (cit. on p. 12).

[19] Herb Sutter. *A Fundamental Turn Toward Concurrency in Software*. https://www.technologyreview.com/2016/03/23/8768/intel-puts-the-brakes-on-moores-law/. Last Accessed: 2021-03-18. MIT Technology Review, 2005 (cit. on p. 1).

[20] Violat Issue Tracker. *IRangeError: Chance: No valid entries in array weights · Issue 10 · michael-emmi/violat*. https://github.com/michael-emmi/violat/issues/10. Last Accessed: 2021-03-16 (cit. on p. 25).

[21] Willem Visser, Corina Pasareanu, and Sarfraz Khurshid. "Test input generation with Java PathFinder". In: *ACM SIGSOFT Software Engineering Notes*. Vol. 29. July 2004, pp. 97–107 (cit. on pp. 12, 41).

[22] Michael Whittaker. *Linearizability Visualizer*. https://github.com/mwhittaker/linearizability. Last Accessed: 2021-02-10. 2015 (cit. on p. 4).

Appendix A

Artifacts

We provide the source code of our plugin, the plugin delivered as a compressed file in the JetBrains marketplace and the code to reproduce the experiments we run in open source repositories using our plugin.

A.1 Artifact Check-list

1. **Source code**:

   We can build the project from source by opening the source code in IntelliJ and running the **Run Plugin** Gradle task. We can find the source code in the repository available in the following link:

   https://github.com/alaukiknpant/intellijViolatPlugin.

2. **Plugin in Marketplace**:

   The plugin has passed all of JetBrains' verification requirements and is published in JetBrains marketplace in the following link:

   https://plugins.jetbrains.com/plugin/16397-violatintegration.

3. **Reproducible Experiments**:

Once the plugin has been downloaded in our IntelliJ IDE, we can reproduce the results presented in the experiments mentioned in Section 3.2. The repository associated with the experiments can be found in the following link:

`https://github.com/alaukiknpant/usingViolatIntegration`.

A.2 Requirements to Run the Artifacts

1. Node.js runtime for JavaScript: version 10.0 or greater
2. Java SE Development Kit: version 8
3. Gradle build tool: at least version 6
4. Maven project management tool
5. Java Pathfinder(JPF) available in your executable

More information on how to add JPF as an executable can be found in the following link:

`http://javapathfinder.sourceforge.net/Running_JPF.html`.

Appendix B

ViolatIntegration Tutorial

We present a tutorial showing a step-by-step process to test the `QueueWrong` class shown in Figure 3.1.

B.1 Installation of ViolatIntegration

There are two ways to install the plugin. In the **Settings/Preferences** dialog , we select **Plugins** and search for **ViolatIntegration** and click install. Alternatively, the plugin archive is also available in the JetBrains Marketplace as a ZIP file.[1] After downloading the ZIP file, the user must again navigate to **Settings/Preferences** dialog , select **Plugins** and click on the **Settings** button and then finally click **Install Plugin from Disk**. After this, the user must select the archive file and they are ready to detect linearizability violations in the IntelliJ Platform .

B.2 ViolatIntegration Settings

Once the plugin has been installed, the user has to provide a valid path to their Violat installation using the **Violat | Settings** tab in their menu

[1] https://plugins.jetbrains.com/plugin/16397-violatintegration

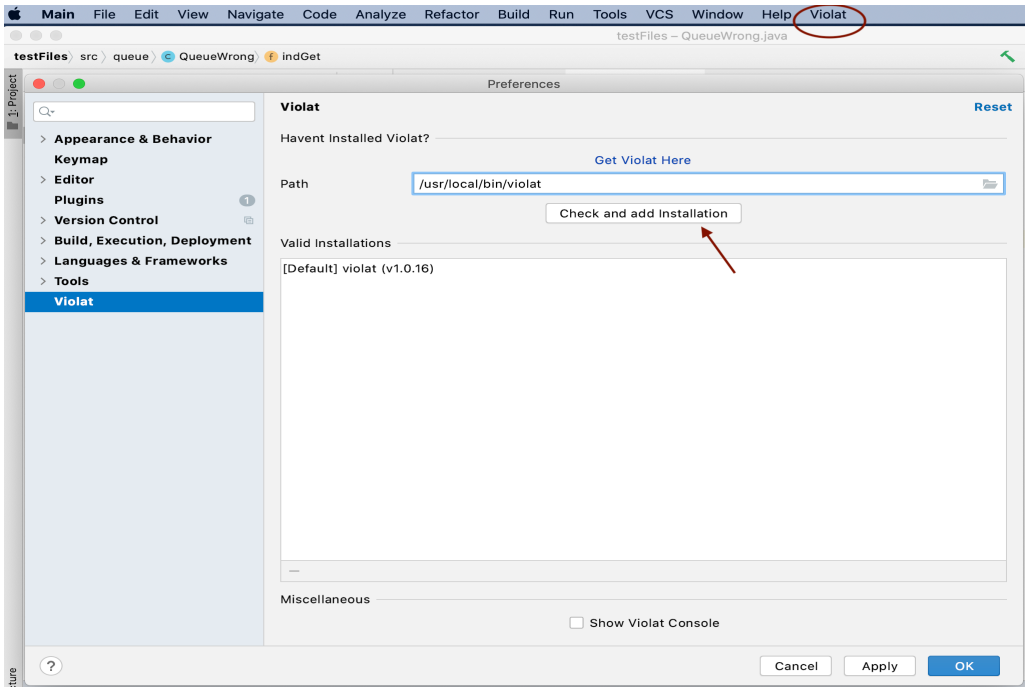

Fig. B.1. ViolatIntegration's Setting.

bar as shown in Figure B.1. If the user does not have a valid Violat installation, the plugin also comes with a link that guides the user to Violat's source code.

B.3 Run Configuration

IntelliJ IDEA utilizes run configurations in order to represent a set of startup properties for running an application. Before detecting the violations associated with the *QueueWrong* class, we need to specify a run configuration for Violat. A Violat Run Configuration can be added by navigating to **Run** | **Edit Configurations** in the menu bar at the top of the screen and choosing **Violat** via the **+** button. After following these steps, a window similar to the one shown in Figure B.2 appears. The user should keep an eye for Run Configuration Errors such as the one shown

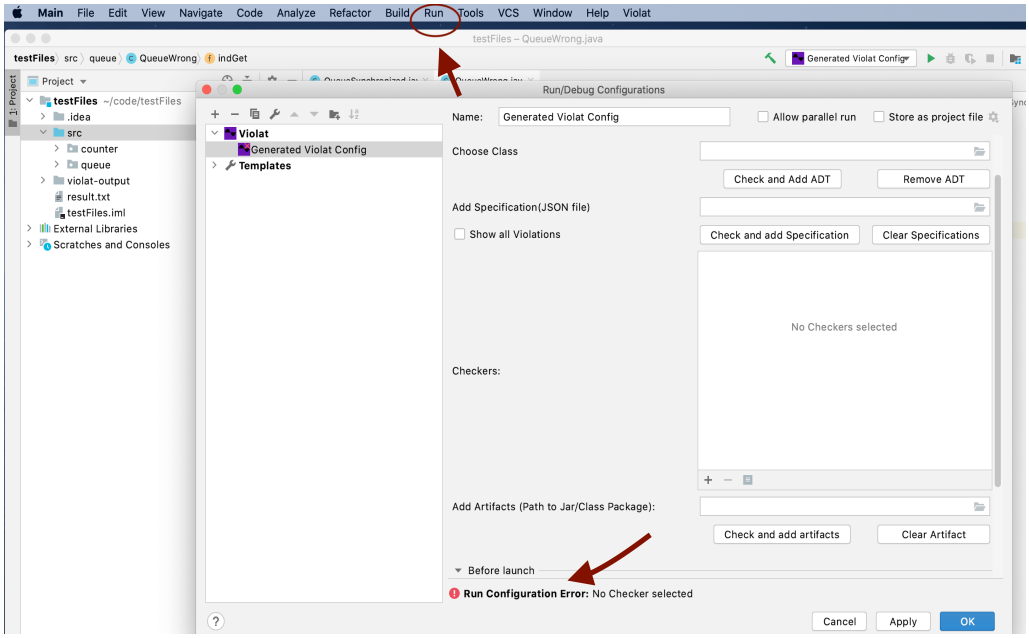

Fig. B.2. Run Configuration.

in Figure B.2, where we get an error because no *Checker* has been selected. Now that we have opened the Violat Run Configuration Editor, we need to select our implementation of the *QueueWrong* class that we want to test. We can do so by clicking on the folder icon in the text field with a browse button next to the *Choose Class* tag, as pointed by the red arrow in Figure B.3. Then, a window showing the file structure of our project appears, from which we can select the *QueueWrong* class. After selecting the relevant class, the name of the class along with its package name appears in text field. At this point, we will enumerate the the next configurations that the user must set up, which is also illustrated in Figure B.4.

1. **Violat Installation**: The user can choose which installation they want to use out of all the installations of Violat that they have added.

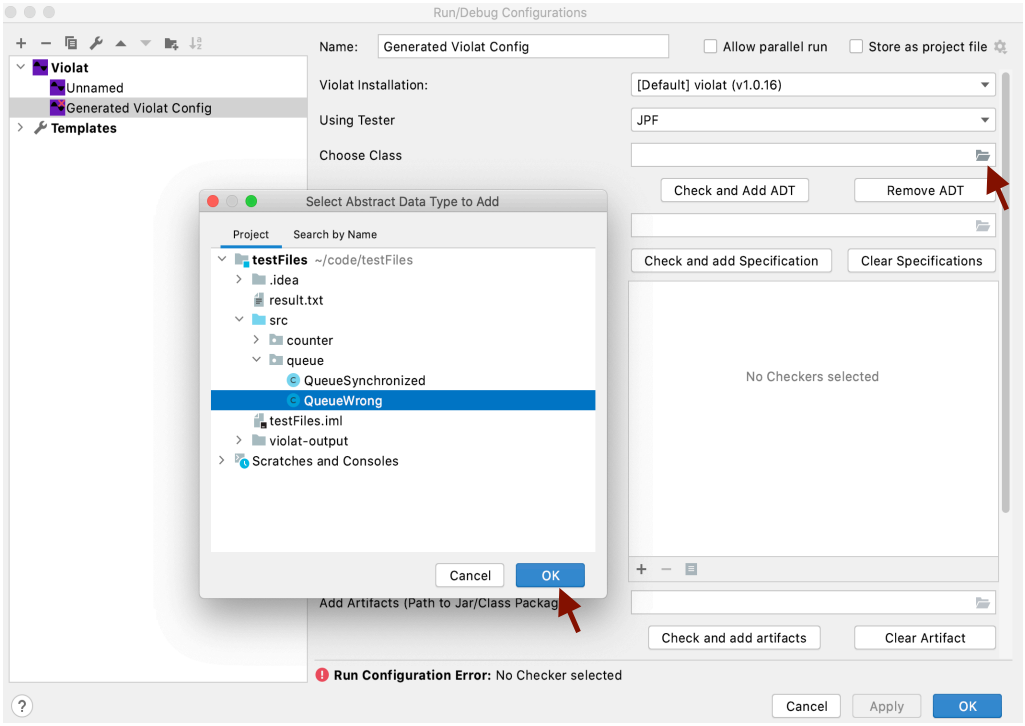

Fig. B.3. Run Configuration when choosing a class.

2. **Tester**: There are two testers that are integrated into Violat. We can use either of the testers, although JPF is the recommended tester.

   (a) JPF (Java Pathfinder): JPF is a system developed by NASA used to verify executable Java bytecode programs for several purposes. In our case, Java Pathfinder exhaustively explores program paths via partial-order reduction and finds consistency violations, and thus linearizability violations [4, 21].

   (b) JCStress (Java Concurrency Stress testing tool): JCStress is used to produce a number of tests to stress test program paths to find consistency violations, and thus linearizability violations. JCStress is not the recommended tester.

3. **Check and Add ADT Button**: This button checks if the selected

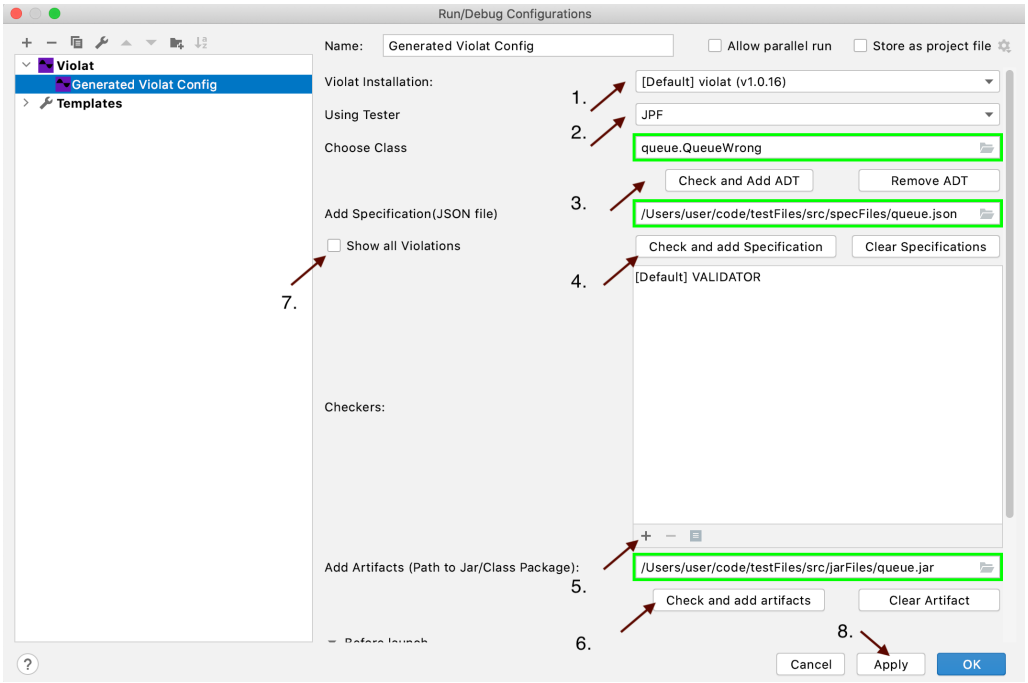

Fig. B.4. Run Configuration when selecting necessary fields.

implementation of the ADT is valid. If yes, it creates a specification (JSON file) and artifact (JAR file) associated with the ADT implementation and adds the path of the generated files in the relevant text boxes. The text field placed above the *Check and Add ADT Button* is highlighted in green if the entire process is successful and in red otherwise. Note that this process might take a bit of time as it involves generating an artifact and a specification.

4. **Check and Add Specification Button**: If we press this button, we can add the aforementioned generated specification (JSON file) describing our ADT, or choose a specification we wrote ourselves or edit the generated specification and add it to our Run Configuration. The associated text field will be highlighted in green if successful and in red otherwise.

5. **Add Checkers**: We must select exactly one checker using the **+** sign. The possible options for checkers are:

   (a) *VALIDATOR*: If we select this checker, Violat gives us executions that that do not yield the predicted outcomes, signaling a violation to linearizability.

   (b) *HISTORIES*: This checker generates figures relating to checking linearizability violations. Unfortunately, this feature was not well tested in Violat for user defined ADTs. Hence, we add a *"[IN PROGRESS]"* tag next to it in anticipation that this feature will be fixed in future releases of Violat.

6. **Check and Add Artifacts Button**: If we press this button, the path to the aforementioned generated artifact (Jar file) or an artifact we specify ourselves is added to the Run Configuration. The text field associated with artifacts will then be highlighted in green if successful and in red otherwise.

7. **Show all Violations Checkbox**: It can take a while to uncover all violations via Violat as the testers can explore a large number of program paths. Hence, not checking this box signals Violat to test only 1 program path. Not checking this box can yield violations faster, but can also give us less violations or incorrectly signal that there are no violations.

8. **Apply Button**: This button just saves the run configuration that will be applied to Violat. We press this button once we are done with all the aforementioned fields.

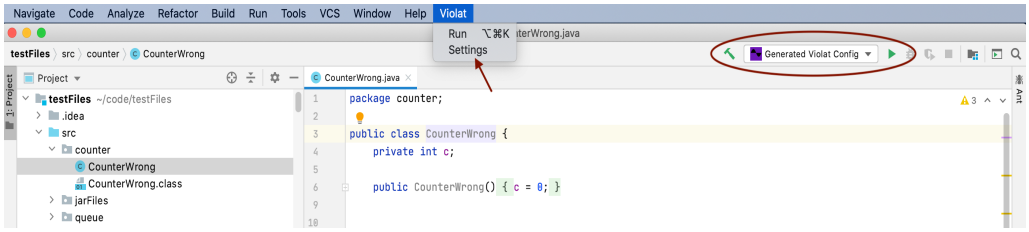

Fig. B.5. Different ways of running ViolatIntegration.

B.4 Getting and Interpreting the Results

Once we have set up the Run Configuration in the manner mentioned above, we can get the results associated with our *QueueWrong* Class by going to **Violat** | **Run** or clicking the play button next to the sign that shows the name of our generated Violat Run Configuration, as shown in Figure B.5. For the *QueueWrong* Class, we get a total of 45 violations, shown in Figure 3.2.