VeriLin Manual

Qiaowen Jia

September 2022

This manual describes how to use the linearizability checker VeriLin. VeriLin is developed in Java and can be used to test whether a concurrent program written in Java is linearizable from its execution traces. We assume the reader is familiar with Java and has a basic understanding of the concept of linearizability. VeriLin supports checking of both simple concurrent objects as well as large-scale concurrent programs.

1 Overview

VeriLin is a linearizability checker that can be applied to large-scale concurrent objects. The algorithm used in VeriLin is based on that of Wing & Gong [1], upon which is added several optimizations and testing techniques to improve efficiency and scalability. After running on a concurrent object, VeriLin will return whether the concurrent object is (believed to be) linearizable or non-linearizable. If the result is non-linearizable, VeriLin will also return a non-linearizable history to the user. VeriLin provides two usage modes that are suitable for different requirements: manual mode and automatic mode. In the manual mode, VeriLin provides a basic framework for linearizability checking, but allow the user to fill in detailed specifications and code instrumentation. In the automatic mode, with either the default or user-provided configuration file, VeriLin can be run fully automatically.

In both usage modes, the execution of VeriLin consists of two stages: generating history and checking linearizability. In the generating history stage, VeriLin invokes the methods in the external API of the concurrent object, either in arbitrary order or according to user-provided scheduling settings (see Section 2). Execution of the methods are done in a concurrent manner, and so can interleave with each other in a way that is determined by the Java scheduler. This results in a trace consisting of invocation and return events of method calls. Next, the linearizability checking algorithm is applied to this history, which tries to find a sequential ordering of the method calls that is consistent with the partial-order in the current history, and whose results of method calls agree with that of the history. If such a sequentual ordering is found, the history is declared as linearizable. Otherwise it is non-linearizable. This process is repeated many

times, and terminates until either a non-linearizable history is found, or if a pre-determined number of runs is reached.

2 Manual Mode

Manual mode is designed for more general testing scenarios, where the user has more control over the specification and other aspects of how linearizability checking is done. In this mode, VeriLin provides a basic framework of generating history and checking linearizability, and users can customize the framework according to the particular requirements of the testing scenario.

2.1 Specifying the Object to be Tested

The following parts are in general needed for specifying the concurrent object to be tested.

- The format for printing each operation (method call) in the history, and the printing and parsing function for that format. The history is printed explicitly during the generating history stage, and serves as the intermediary between the generating history and checking linearizability stages. Any printing format should include the following information for each operation: **preTime**, **postTime**, **threadId**, and **operationName**. For example, for a lock-free queue, the printing format for an operation is as follows:

  preTime postTime threadId operationName value result

  Here **operationName** is one of `enq` (for enqueue) and `deq` (for dequeue). Field **value** is the argument to *enq* or the return value of *deq*. Field **result** shows whether the operation is successfully executed.

- Scheduling Settings. For some concurrent objects, methods in its external API can be called in any order. In this case, fully random scheduling of method calls is appropriate. However, for many real-world objects, methods must be called in a particular order, for example as specified by an automaton. The manual mode allows providing the scheduling settings that are suitable for the concurrent object to be tested.

- Identification of the specification, external methods and their properties. The specification and external methods should be declared in the setting for VeriLin. The properties of methods include whether it is reversible, and whether it is read-only or may update the state of the object. These properties will be used in the linearizability checking procedure to achieve certain optimizations.

- Exception/special data handling when executing methods in the concurrent object to be tested, or in the specification object. In practice, implementations of these objects vary, so users should deal with exceptions and special data cautiously, and guarantee these cases correspond to the specification.

2.2 Code Instrumentation

Code instrumentation allows inserting additional printing information into the concurrent object to be tested, as well as adjusting the invocation/return point of the methods. One use of instrumentation for providing additional information is as follows: suppose the return type of all external methods of the current object is void, then the default printing format of the operations cannot expose sufficient information. In this case, the user can instrument the code for the concurrent object in order to print more information about the state of the object. The position of such instrumentation is usually after an important event or after an implicit function is executed.

2.3 Adjusting Waiting Time

In the stage of generating history, the user can adjust the waiting time before, after, or during the execution of an operation. This allows some degree of control between the amount of interleaving in the resulting history, in particular the size of *regions* in the history, and achieve a balance between testing time and amount of concurrent execution that is tested. Noted that the position of waiting is located between the reported **pretime** and **posttime**. The waiting time can be specified with milli-seconds or nano-seconds as units. With Verilin, waiting time is implemented by invoking `Thread.sleep()` in Java.

2.4 Example: the Train Ticketing System

We now introduce the train ticketing system as a large-scale example to demonstrate how to use VeriLin in the manual mode.

First, we give a brief introduction to the train ticketing system. The system supports clients to buy or refund a ticket for a given route, departure station and arrival station, as well as to inquire the number of tickets remaining given the same inputs. To use VeriLin in the manual mode, we integrated it into the `ticketingsystem` Java package. There are four main files in the `ticketingsystem` package:

- `TicketingSystem.java` defines the interface of train ticketing system.
- `TicketingDS.java` is a simple implementation of the train ticketing system and is used as the specification.
- `GenerateHistory.java` is the test-case generation program for VeriLin.
- `VeriLin.java` is the main linearizability checking program.

The input parameters provided to `GenerateHistory` are as follows:

- **threadNum** is the number of threads to run the implementation.
- **operationNum** is the number of operations involved in each thread.
- **isSequential** is a boolean value that specifies whether testing is done in sequential or concurrent mode. In the sequential mode, only one operation is running at any time. In the concurrent mode, multiple operations can run at the same time.
- **msec** is the number of milli-seconds to delay before calling a method.
- **nsec** is the number of nano-seconds to delay before calling a method.

Given the input parameters, `GenerateHistory` calls the buy/refund/inquiry methods of the implementation in a random manner. The history generated from the implementation is recorded into a history file by `GenerateHistory`. Next, `VeriLin` sorts and parses the history, performs linearizability checking using `ticketingDS` as the specification. The input parameters provided to `VeriLin` are as follows:

- **threadNum** is the number of threads to run the implementation with.
- **historyFile** is the history file generated by by `GenerateHistory`.
- **isPosttime** is a boolean value. If it is true, then VeriLin sorts the operations by their response time. If it is false, then VeriLin sorts the operations by their invoke time.
- **failedTrace** is the file that contains a non-linearizable history of minimal length that is found by VeriLin (if any).

When VerLin finished the checking procedure, if it finds the history to be linearizable, it prints the following message:

```
history size = 2916, region size = 2804, max_region_size = 63
Verification Finished. 33ms
```

If it finds the history to be non-linearizable, it prints the following message:

```
history size = 2947, region size = 2797, max_region_size = 14
Verification Failed. 197ms
```

It will then generate a file `failedHistory_026_221` containing the non-linearizable history of minimal length that is found.

There are several common settings of input parameters.

1. **Default Mode.** In the default mode, the parameter to `GenerateHistory` is **threadNum operationNum** 0 0 0. In this mode, history is generated concurrently, and VeriLin uses **postTime** for initial sorting of the operations by response time. All methods are treated equally.

2. **Replay Mode.** In the replay mode, the parameter **isSequential** is set to 1. In this mode, history is generated sequentially, so that each region has only one operation. If an error is found, VeriLin can replay the history according the specification, with no backtracking step executed. User can set breakpoints in `VeriLin.java`, indicating at which line of the history file will the replay be stopped. It is very useful when testing correctness of the specification itself.

3. **Delay Mode.** In the delay mode, user provides the parameters **msec** and **nsec**. Then `GenerateHistory` will delay for a specific amount of time before calling a method, controlled by the **msec** and **nsec** parameters.

4. **preTime mode**

   VeriLin offers the preTime mode as an option for sorting the operations by their invoke time. The parameter isPosttime needs to be false for this mode. For most train ticketing system implementations, the performance of the postTime (response time) mode is better than that of the preTime mode.

3 Automatic Mode

The automatic mode allows fully automatic linearizability checking, after providing the corresponding configuration file (or choosing to use the default configurations). In the automatic mode, the user should provide three files: the Java file containing the concurrent object to be tested, the Java file containing the specification of the object, and the configuration file. If the configuration file is valid, VeriLin can generate code for the project and check linearizability automatically.

In this mode, the configuration file is essential. It includes several customized properties with respect to the concurrent object to be tested and the corresponding specification. The configuration file below shows an example of the configuration for a list implementation.

```
ObjectName = testlist;
FileSuffix = java;
FileType = selfDefined;
SpecFlag = true;
SpecFileName = list_int;
MethodList: {insert, delete, contains};
MethodFreq: {30, 30, 50};
SpecRelation: {insert, add};
SpecRelation: {delete, remove};
ReversibleRelation: {add, remove};
ReversibleRelation: {remove, add};
ReversibleRelation: {contains, contains};
```

Explanation for the fields in the configuration file are as follows.

- *ObjectName* and *FileSuffix* compose the whole file name of the concurrent object to be tested.

- *FileType* has two values: `selfDefined` and `preDefined`. `preDefined` means the concurrent object to be tested is integrated within VeriLin, and the user needs to identify the name of the `preDefined` object. If the *FileType* is `selfDefined`, it means that the object is given by the user in the same path.
- *SpecFlag* determines whether the specification object is given by the user. If *SpecFlag* is set to true, the specification has been given by the user, otherwise VeriLin will take the object to be tested as the specification object.
- *SpecFileName* declares the file name of the specification.
- *MethodList* is the set of external methods of the object. Generally, the method list contains only methods of the upper layer.
- *MethodFreq* has the same length as the *MethodList*. It declares the frequency of appearance of each test method during random scheduling.
- *SpecRelation* provides the relation between the methods of the object to be tested and methods from the specification. For example, in the object testlist to be tested, the method insert corresponds to the method add in the specification of a list.
- *ReversibleRelation* declares the reversible relation of the methods to be tested. For example, in the non-empty stack, push and pop is a pair of reversible methods. Each direction of the reversible relation should be included in the configuration.

The automatic mode uses an external package called JavaParser to help parse the given Java files. In this mode, the configuration file with the format described above must be named as `config`, and the files containing the concurrent object to be tested and its specification should be located in the same path. The source files for generating a linearizability tester is located in `src/main/java/`, where the source files in `AutoGenerator/` aim to generate `GenerateHistory.java` and `VeriLin.java` in `verify/`. The file `history` containing the generated history, and the file `failed_history`i containing a non-linearizable history in the i-th test, are also located in the directory `verify/`. The generated code needs to be built with the auxiliary files in other folders in `src/main/java/`.

We provide a push-button shell script `test.sh` to execute all these steps automatically: first generating `GenerateHistory.java` for collecting histories and `VeriLin.java` for checking their linearizability, then building and running the generated code to get the test result. For example, checking the linearizability of a list object of integer elements is executed as: test.sh list list_int.

Currently the automatic mode is still a prototype, and not a release version. Some errors may still occur during the checking process for so far undetermined reasons. For instance, when a collection is empty, the handling method is not

unique: they may raise exception, return -1, 0 or `Integer.MIN_VALUE`. Future work should be done to deal with these special cases.

4 Setting the Specification

There are three ways to set the specification of the concurrent object to be tested: built-in, user given, and taking the sequential execution of the object itself as the specification. If the third way is chosen, VeriLin provides an extra replay and debugging stage to check the correctness of the sequential history of the object. It should be noted that replay and debugging can only be applied in the manual mode. A pre-condition for using the replay mechanism is that the user also provides some constraints of the object to be tested, which will be checked at runtime.

Currently, VeriLin provides five built-in specifications: List, Set, Queue, Stack and Double-ended queue (Deque). All five specifications are derived from corresponding data structures in the Java Standard Library.

References

[1] Jeannette M. Wing and Chun Gong. Testing and verifying concurrent objects. *Journal of Parallel and Distributed Computing*, 17(1-2):164–182, 1993.