

JavaBIP meets VerCors: Towards the Safety of Concurrent Software Systems in Java (Artefact README)

Simon Bliudze Petra van den Bos Marieke Huisman
Robert Rubbens Larisa Safina

Contents

Summary	1
Quick start	2
Artefact completeness caveat	2
Paper abstract	2
General design of the artefact	3
General structure of the artefact	3
Terminology overview	4
Setup	4
Hardware requirements	5
Test Instructions	5
Replication Instructions	6
Dynamic detection	6
Static detection & verification report	7
Dynamic checking optimization	8
Adjusted contract checking	8
Examples of Usage	9
License	10

Summary

This readme discusses the contents of this artefact, and instructions on using it. The `.pdf` and `.md` files are identical.

Quick start

To quickly test the artefact, first put the files in a folder in the FASE'23 VM, acquired from here: <https://doi.org/10.5281/zenodo.7446277>. Then, run the following scripts in the FASE'23 VM in the following order, no internet connection necessary:

- `22-runJavaBipAdjustedCasino.sh`: logs execution of the model. Does not crash, might print about violated invariants and freeze, otherwise runs indefinitely.
- `30-installVerCors.sh`: terminates after a few seconds. Prints `VerCors 2.0.0-javabip-alpha`.
- `32-runVerCorsCasinoAdjusted.sh`: terminates after a while, prints `Verification successful`. Produces `outputs/actualReportAdjusted.json`, containing only “`proven`” entries.

Any significant deviations indicate a possibly broken artefact.

Artefact completeness caveat

The implementation included with this artefact contains a bug. Two days before the artefact deadline we discovered a bug in the execution of the Casino model that we have not been able to resolve since. We suspect that it might be either an edge case triggered by the runtime verification functionality implemented in JavaBIP, or a bug in the deductive verification functionality that causes some behaviour of the Casino model to be excluded from analysis. Therefore, the “Casino Adjusted” case study, in the replication scenario of **Adjusted contract checking**, produces violated invariants. In our understanding of the model, these violations should not happen.

In our opinion, the technique presented in our paper is sound. Moreover, we think this bug is a technicality, and not a flaw in the general approach. Finally, because the artefact is otherwise fully functional and carefully documented, we include it with our paper.

Paper abstract

We present “Verified JavaBIP”, a tool set for the verification of JavaBIP models. A JavaBIP model is a Java program where classes are considered as components, their behaviour described by finite state machine and synchronization annotations. While JavaBIP guarantees execution progresses according to the indicated state machine, it does not guarantee properties of the data exchanged between components. It also does not provide verification support to check whether the behaviour of the resulting concurrent program is as (safe as) expected. This paper addresses this by extending the JavaBIP engine with run-time verification support, and by extending the program verifier VerCors to verify JavaBIP models deductively. These two techniques complement each other: feedback from run-time verification allows quicker prototyping of contracts, and deductive

verification can reduce the overhead of run-time verification. We demonstrate our approach on the “Solidity Casino” case study, known from the VerifyThis Collaborative Long Term Challenge.

General design of the artefact

The contents of this artefact can be divided into four categories:

- Tool sources: JavaBIP engine & VerCors verification tool sources
- JavaBIP Casino case study sources
- Binaries: binaries of the JavaBIP & VerCors tool, compiled from the included sources
- Shell scripts: scripts to install dependencies required for compilation, compile the sources into binaries, install the binaries in the FASE’23 VM, and run the case study.

The primary goal of this artefact is reproducing the results in the paper. For this only the FASE’23 VM, the JavaBIP Casino case study sources, binaries, and a subset of the shell scripts are necessary. The other folders and shell scripts can be ignored. No internet connection is needed for this. The authors expect this part of the artefact to be robust and to be functional for a long time. This readme contains instructions on everything related to achieving this goal.

The secondary goal of this artefact is gathering sources and compiling the binaries needed to reproduce the results in the paper. If you are only interested in reproducing the results, you can ignore the files required for this part. For this goal the FASE’23 VM is needed, *as well as an active internet connection (!)*, the sources, and most of the shell scripts. In addition, this process is not just dependent on the VM, but also on source repositories and external packages being available. For example, the VerCors repository, Maven Central, and debian repositories accessible via `apt`. This part of the artefact is less robust and might break over time as urls change and repositories disappear. However, the reason the authors included this infrastructure was because it will likely work for quite some time, and could be useful to parties interested in further extending our tools. Documentation related to this goal can be found in the `AUTHOR_README.pdf` file.

General structure of the artefact

Files and folders relevant for reproduction:

- `binaries/`: contains precompiled binaries of the JavaBIP engine & VerCors, ready to be installed. These `binaries` are compiled ahead of time by the authors from the sources in the `sources` directory.
- `sources/`: contains sources of the four projects needed to compile & run the Casino case study discussed in the paper.
- `outputs/`: contains the expected verification reports, as well as the verification reports when they are generated by VerCors.

- `20-runJavaBipBrokenCasino.sh`, `21-runJavaBipBrokenCasinoWithReport.sh`, `22-runJavaBipAdjustedCasino`, `23-runJavaBipAdjustedCasino.sh`: Scripts to run JavaBIP and the Casino case study. If these scripts do not encounter an error during execution, they will run indefinitely. The “WithReport” versions depend on a verification report being generated by VerCors first!
- `30-installVerCors.sh`, `31-runVerCorsCasinoBroken.sh`, `32-runVerCorsCasinoAdjusted.sh`: Scripts to install & run VerCors and the Casino case study. Should all terminate within a minute or so.
- Scripts 00 - 13: these are for packaging the artefact, and can be ignored for reviewing. For more info, see `AUHTORS_README.pdf`.
- `README.md`, `README.pdf`: This readme.
- `LICENSE`: Contains the license for the content of this artefact.

All other files and folders are only relevant to compiling the tools into binaries. See `AUTHOR_README.pdf` for details.

Terminology overview

- “Broken casino”: this refers to the version of the casino case study that contains a bug. Specifically, the operator tries to withdraw more money than is present in the casino.
- “Adjusted casino”: this refers to the version of the casino case study that has its runtime guards adjusted to ensure that the transition that triggers this bug is never enabled. This change fixes the bug, at the cost of introducing a deadlock. Because a deadlock can also be interpreted as a bug, we named this version the “adjusted” version, instead of the “fixed” version.
- “Report”, “verification report”: this is the name of the file that VerCors produces after deductive verification of a JavaBIP model. It contains for each component which contracts have been verified, and which contracts it was not able to verify. Using this report, the JavaBIP Engine can at runtime choose to skip checking contracts that have already been verified by VerCors. Note that this file is distinct from the textual output that VerCors produces in the terminal when executed.

For explanation of other concepts discussed in this readme, we refer the reader to the full paper “JavaBIP meets VerCors: Towards the Safety of Concurrent Software Systems in Java”.

Setup

First, put all the files in a folder in the FASE 2023 VM. The VM can be acquired from here: <https://doi.org/10.5281/zenodo.7446277>.

To install VerCors run the `20-installVerCors.sh` script. This will put the `vercors` executable in the `PATH`.

JavaBIP is included with this artefact as pre-packaged executable JARs

Hardware requirements

The artefact should work on any recent laptop that can reasonably run the FASE'23 VM. There are no specific hardware requirements.

Test Instructions

Script order & dependencies

- `20-runJavaBipBrokenCasino.sh` and `22-runJavaBipAdjustedCasino.sh` can be executed immediately.
- `31-runVerCorsCasinoBroken.sh` and `32-runVerCorsCasinoAdjusted.sh` depend in VerCors being installed first with `30-installVerCors.sh`
- `21-runJavaBipBrokenCasinoWithReport.sh` and `23-runJavaBipAdjustedCasinoWithReport.sh` depend on first running the above VerCors scripts, as the VerCors scripts produce a verification report. The report is generated in the `outputs/` folder.

VerCors

To test if VerCors was installed properly the version can be queried:

```
$ vercors --version
VerCors 2.0.0-javabip-alpha
```

To test if VerCors can run properly on a small test input, a test file can be verified:

```
$ vercors sources/vercors-javabip/examples/concepts/goto/goto2.pvl
[INFO] Starting verification
[INFO] Verification completed successfully.
```

A progressbar should briefly be visible. Warnings about viper reparsing or logging configurations are benign and can be ignored.

JavaBIP

To test if JavaBIP is properly included, run the first JavaBIP script:

```
$ ./20-runJavaBipBrokenCasino.sh
... model events logging ...
```

The execution might run indefinitely. It might also signal a violated invariant by printing `Runtime verification exception`, with the violated invariant printed before that. The execution can be terminated manually by pressing (Ctrl+C).

Replication Instructions

There are four usage scenarios to evaluate:

1. The JavaBIP engine dynamically detects a contract violation. (**Dynamic detection**)
2. VerCors statically detects a contract violation, and produces a verification report. (**Static detection**)
3. When including the verification report, JavaBIP omits checking contracts that will not be violated. (**Dynamic checking optimization**)
4. After adjusting the contracts, neither JavaBIP nor VerCors detects a contract violation. In addition, JavaBIP omits checking any invariants when the verification report is included. (**Adjusted contract checking**)

The “contract violation” in the case study is that more money can be withdrawn from the pot than is available, as discussed in the paper. We will now discuss how to replicate each of the four scenarios, up to the caveats as described at the beginning of this document. It is assumed that the setup steps from the “Setup” section have been performed.

Dynamic detection

Run the `20-runJavaBipBrokenCasino.sh` script. This will run the broken casino case study in the JavaBIP engine. Execution is not deterministic, so it might take several tries to get a contract violation. At some point the terminal may prompt some of the following messages on the contract violation and terminates while printing the violated invariant. We included an example of this below, with added newlines for presentation purposes:

```
... Log prefix omitted ...
OPERATOR101: decided to withdraw 40, wallet: 161
CASINO201: GAME CREATED
OPERATOR101: making one step in the game
PLAYER301: bet 146 placed, purse: 44
CASINO201: received bet: 146, guess: HEADS from player 301
PLAYER301: won 292 purse: 336
CASINO201: 146 lost, pot: 25
OPERATOR101: making one step in the game
OPERATOR101: making one step in the game
CASINO201: GAME CREATED
PLAYER301: bet 141 prepared, purse: 195
CASINO201: 40 removed by operator 101, pot: -15
09:29:05.802 [ACTOR_SYSTEM-akka.actor.default-dispatcher-2]
    ERROR org.javabip.executor.BehaviourImpl - Casino: Invariant violation:
    secretNumber != null && bet >= 0 && pot >= bet
09:29:05.802 [ACTOR_SYSTEM-akka.actor.default-dispatcher-5]
    ERROR org.javabip.executor.BehaviourImpl - Operator: State predicate violation:
```

```
0 <= amountToMove && amountToMove <= pot
for the state: WITHDRAW_FUNDS
```

Log explanation

On the first line, the operator decides to withdraw 40. Then a game takes place, which leaves the casino with a pot of 25. Finally, the operator and casino synchronize to actually withdraw 40, which is broken down into two steps. First, the deduction from the pot takes place in the casino component, after which the casino invariant cannot be re-established. Second, the deduction from the pot takes place in the operator component, which causes a precondition to be violated. Execution freezes after the violated invariants and conditions.

Static detection & verification report

Run the `31-runVerCorsCasinoBroken.sh` script. This will check the broken casino case study with VerCors. Verification can take up to 20 seconds, depending on the computer. A progress bar should be visible in the mean time.

Five errors are printed, indicating that the pot balance might become negative. They are printed in verbose mode, so it should require some scrolling.

We will briefly explain the first error. For each error in the output from VerCors, three source code parts are highlighted:

1. The specific transition causing the error. This is important because each transition can have its own specific preconditions, which might influence the proof of correctness of the method. In this case, it is the first `@Transition` annotation.
2. The method the transition is associated to, for completeness. This is `removeFromPot` from `Casino.java`
3. The part of the invariant that does not hold. In this case, this is `pot >= bet`.

In addition, the component invariant also contains a constraint that `bet >= 0`. Meaning, VerCors cannot establish that pot will remain non-negative after removing funds from the pot.

A verification report will also have been produced by this verification run in `outputs/actualReportBroken.json`. In this file, most transitions will only have “**proven**” entries. The five transitions of which the postcondition is not verified, will have a “**not proven**” entry. For example, we can look up the results for the transition error discussed above. Its has the following signature, according to the `@Transition` annotation:

- Port: `REMOVE_FROM_POT`
- Source state: `IDLE`
- Target state: `IDLE`
- Guard: `IS_OPERATOR`

Looking that combination up in the report, we find the following block:

```
"signature": {
  "name": "REMOVE_FROM_POT",
  "source": "IDLE",
  "target": "IDLE",
  "guard": "IS_OPERATOR"
},
"results": {
  "precondition": "proven",
  "componentInvariant": "not proven",
  "stateInvariant": "not proven",
  "postcondition": "not proven"
}
```

To summarize, because of the non-verifiable component invariant in that transition all three of the contracts (component invariant, state invariant, and postcondition) could not be proven. Hence, these will need to be checked at runtime.

To doublecheck that the verification report does not deviate from the expected result, it can be compared to the file `outputs/expectedReportBroken.json`.

Dynamic checking optimization

Assuming the verification report was produced by VerCors, the checking done at runtime by the JavaBIP engine can be optimized by omitting the checks for contracts that are proven to never be violated. To observe this, first run `31-runVerCorsCasinoBroken.sh` to produce a verification report in `outputs/actualReportBroken.json`. Then, run `21-runJavaBipBrokenCasinoWithReport.sh`, which uses the report produced earlier. JavaBIP logs that the verification report is now used, after which only the invariants not proven by VerCors are checked.

Adjusted contract checking

When the contract violation is no longer present in the model, 1) VerCors should no longer detect violated contracts, and 2) the JavaBIP engine should not detect any contract violations at runtime. We have applied the necessary changes to the “casino broken” case study to fix all contract violations in the “casino adjusted” case study. The differences between the broken and adjusted casino case studies are few. They are listed in the `casinoCaseStudyDiff.txt` file.

Unfortunately, because of a technicality in the implementation, the artefact does not properly demonstrate this. For more information, please see the completeness caveat at the beginning of this document. For completeness, we do explain below the structure of this replication scenario, even though the results are not as expected from a correct model.

VerCors

To check the adjusted casino case study with VerCors, run the `32-runVerCorsCasinoAdjusted.sh`. This will print “**Verification completed successfully**”, indicating that no contracts are violated. In addition, the file `outputs/reportAdjusted.json` will be generated, containing only “**proven**” entries.

JavaBIP

To run the adjusted casino case study with JavaBIP, run the `22-runJavaBipAdjustedCasino.sh`. This should keep running indefinitely, as no contract violations will be detected. However, because of a bug in the model or implementation, a violated invariant will likely take place.

If `32-runVerCorsCasinoAdjusted.sh` was executed before, the `23-runJavaBipAdjustedCasinoWithReport.sh` script can be executed. This executes JavaBIP on the adjusted casino case study, including the verification report. This causes none of the invariants to be checked, which is safe to do, as they have all been proven to hold. Because of the changes made to the adjusted casino case study, the model might go into a deadlock, and hence might stop printing output without an error message.

This is by design. As described in the paper, the adjusted version of the casino case study has additional runtime guards that prevent the erroneous transition to be taken. This means the bug will never occur. However, in return, a state might be reached where no transition can be taken, causing a deadlock. A proper fix of the bug would require a larger refactoring of the model.

Examples of Usage

Assuming VerCors is installed.

Manually verify a PVL or Java file from VerCors’ included example collection:

```
# Start in artefact root folder
$ vercors sources/vercors-javabip/examples/concepts/goto/goto2.pvl
[INFO] Starting verification
[INFO] Verification completed successfully.
```

Manually verify the broken casino case study, using `*` to glob all input files:

```
# Start in artefact root folder
$ vercors casinoCaseStudyBroken/*.java --more
[INFO] Starting verification
... logging and errors ...
```

Run the casino case study model manually:

```
# Start in artefact root folder, move into the binaries folder
$ cd binaries
# Remove possibly present verification report to ensure entire model is monitored
```

```
$ rm -f verificationReport.json
# Execute model. The classpath is broken down into two lines for presentation purposes,
# but when executing the commands it needs to be on _one line_!
# See the JavaBIP execution scripts for details
$ java -cp \
    org.javabip.examples.casino.broken-0.1.0-SNAPSHOT.jar # Remove line break here!
    :org.javabip.examples.casino.broken-0.1.0-SNAPSHOT-jar-with-dependencies.jar \
    casino.Main
```

Run the casino case study model including the verification report:

```
# Start in artefact root folder, move into the binaries folder
$ cd binaries
# Copy report belonging to this model to the current working directory
$ cp ../outputs/actualReportBroken.json verificationReport.json
# Execute model, it will pick up the verification report from the current working directory
# The classpath is broken down into two lines for presentation purposes,
# but when executing the commands it needs to be on _one line_!
# See the JavaBIP execution scripts for details
$ java -cp \
    org.javabip.examples.casino.broken-0.1.0-SNAPSHOT.jar # Remove line break here!
    :org.javabip.examples.casino.broken-0.1.0-SNAPSHOT-jar-with-dependencies.jar \
    casino.Main
```

License

The code included in this artefact from `javabip-engine`, `javabip-core`, `javabdd`, and `vercors` have own licenses.

All other content in this artefact is licensed under the license in the `LICENSE` file.