# Chapter 6

# Experimental Evaluation of the New Version of Atomer

This chapter is devoted to *testing* and *experimental evaluating* the new version of *Atomer* as proposed and implemented in Chapters 4 and 5, respectively. Each Atomer's enhancement/extension was tested independently as soon as it had been implemented. It was tested on suitable programs created for testing purposes and got inspired by test programs from [17]. Section 6.1 includes the evaluation of analysis *scalability*. Further, Section 6.2 shows experiments performed on test programs derived from the static analyser *Gluon*. Moreover, Section 6.3 is about the experimental evaluation of the new version of Atomer on publicly available *real-life complex* programs. Finally, Section 6.4 concludes the experimental evaluation and discusses future work.

### Testing on Hand-Crafted Examples

The correct behaviour of the first version of Atomer was already tested in [17] on a suitable test suite containing C programs that use *PThread locks*. Test programs from this test suite include sequences of function calls inside and outside *atomic blocks*, *not paired lock/unlock calls*, iteration, selection, and *nested function calls* inside atomic sections. The programs were designed to check all essential aspects of Atomer's analysis, i.e., to check whether various parts of *abstract domains* work well w.r.t. to the proposal.

The implementation of the new version of Atomer was, at first, tested on the test suite from [17]. However, the reference outputs for these test programs were modified to adapt the newly implemented features, e.g., the approximation of working with *sequences of function calls* by working with *sets of function calls* has to be reflected in the reference outputs. Further, new test programs were derived from those in [17], and they were changed in order to check all aspects of the extensions and enhancements implemented in the new version of Atomer. During testing, various kinds of different lock mechanisms for C/C++/Java were used.

This way, the correct functioning of the analysis of Atomer's new version has been validated (w.r.t. the proposal). All of these testing programs are available in the attached memory media; see Appendix A.

## 6.1 Scalability Benchmark

The *scalability* of the analysis has been tested on a subset of a *publicly available benchmark* from [20]. It consists of *real-life low-level complex concurrent C* programs derived from the Debian GNU/Linux distribution. The entire benchmark was initially used for an experimental evaluation of Daniel Kroening's static deadlock analyser for C/PThreads implemented in the CPROVER framework. However, it can also be used for the evaluation of Atomer's scalability. For the evaluation, 7 programs with deadlocks and 54 deadlock-free programs (806,431 lines of code in total) were used.

The experiments[1] were run in a *Docker container* on Windows 10 (WSL 2) with a 3.3 GHz Intel© Core™ i5-2500K 64-bit processor and 6 GB RAM running the Debian GNU/Linux x86-64 10.9 (Buster) operating system. Table 6.1 shows aggregated results of the evaluation. There are average and total times of analyses for both phases of the analysis for both the basic version of Atomer (Atomer v1.0.0, i.e., the version before the *approximation with sets* proposed in the thesis) and the new version (Atomer v2.0.0, i.e., the version after the approximation). Also, there is shown the number of *timeouts* of individual analyses where the timeout $\top$ was set to 10 minutes. It is evident that, on average, the new version of Atomer is about *two times faster*.

Table 6.1: *Aggregated results* of the evaluation of Atomer's *analysis scalability*

| | Atomer v1.0.0 | | Atomer v2.0.0 | |
|---|---|---|---|---|
| | **Phase 1** | **Phase 2** | **Phase 1** | **Phase 2** |
| **Average Time (s)** | 70.98 | 109.11 | 37.96 | 50.93 |
| **Total Time (s)** | 4,117 | 5,892 | 2,164 | 2,750 |
| **Timeouts** | 4 | 9 | 3 | 4 |

For a closer look, Table 6.2 provides an overview of analyses' times of selected programs from the benchmark. Above that, there are `tgrep` and `sort` from GNU Core Utilities that also use PThreads. It can be seen that the approximation decreased the analysis time a lot for some of these programs. Nonetheless, in one case, the timeout was also exceeded in the new version of Atomer. This shows that the analysis may still take a quite long time in some cases even after the approximation. The reason is that in these low-level programs, there are vast and complicated functions (thousands lines of code) that contain *in-lined code*.

## 6.2 Evaluation on Validation Programs Derived from Gluon

*Gluon* is a prototype tool for static analysis of *contracts for concurrency* in Java programs developed in [13, 35] (also discussed in Section 3.1). The functionality of Gluon was validated on a set of small benchmarking programs with known atomicity violations, which can be seen as contract violations. These programs were adapted from [2, 3, 39], where they are typically used to evaluate atomicity violation detection methods. In [13, 35], these

[1] Performed **scalability experiments** are available at GitHub: `https://github.com/harmim/vut-dip/tree/master/benchmarks/scalability`.

Table 6.2: *Individual results* of the evaluation of Atomer's *analysis scalability* ($\top > 600\,\text{s}$)

| Program | LOC | Atomer v1.0.0 | | Atomer v2.0.0 | |
|---|---|---|---|---|---|
| | | Phase 1 Time (s) | Phase 2 Time (s) | Phase 1 Time (s) | Phase 2 Time (s) |
| `btscanner` 2.1 | 9,142 | $\top$ | N/A | 47.33 | 69.04 |
| `daemon` (`libslack`) 0.6.4 | 42,857 | 280.75 | 40.85 | 11.40 | 10.43 |
| `crossfire-client-gtk2` 1.71 | 41,185 | 1.74 | $\top$ | 1.71 | 14.58 |
| `c-icap` 0.4.2 | 39,254 | 154.86 | 20.70 | 2.37 | 6.14 |
| `hmmer2` (`hmmcalibrate`) 2.3.2 | 38,301 | 109.79 | $\top$ | 10.07 | $\top$ |
| `towitoko` 2.0.7 | 12,339 | 38.23 | 77.50 | 2.27 | 10.69 |
| `mesa-demos` 8.3.0 | 2,424 | 397.10 | $\top$ | 46.50 | 5.83 |
| `freecell-solver` 3.26.0 | 15,408 | 37.28 | 0.78 | 3.45 | 9.81 |
| `tgrep` 1 | 3,190 | 240.36 | $\top$ | 0.92 | 0.74 |
| `sort` (GNU Coreutils) | 7,523 | 5.33 | 0.39 | 1.49 | 0.91 |

programs were appropriately redesigned, and the necessary contracts for each program were created.

Moreover, in [13], to validate a *dynamic approach* for contract violations, a subset of the benchmarking programs was rewritten to a C++ version. This subset was also used to validate the new version of Atomer. Validation results[2] are given in Table 6.3. The table shows the number of atomicity violations that can really occur in a given program (*Real Atomicity Violations*), the number of the real violations that were found by the new version of Atomer (*True Positives*), and the number of false positives (*False Positives*).

Table 6.3: Atomer's *validation results* on test programs derived from *Gluon* [13, 35]

| Benchmark | LOC | Real Atomicity Violations | True Positives | False Positives |
|---|---|---|---|---|
| Account [39] | 56 | 1 | 1 | 0 |
| Coord03 [2] | 119 | 1 | 1 | 0 |
| Coord04 [3] | 55 | 1 | 1 | 0 |
| Local [2] | 30 | 1 | 1 | 0 |
| NASA [2] | 97 | 1 | 1 | 0 |

The experiments were performed such that the contracts from [13] were manually given to the second phase of Atomer's analysis as *atomic sets*. As the table shows, Atomer detected *all known atomicity violations* with the *absence of false positives*. Times of the analysis are not included in the table because all the programs were analysed in less than a second.

Furthermore, the author of this thesis created a *correct version* of each benchmark, i.e., a version where atomicity violations are fixed by adding appropriate locks. Then, it was shown that the first phase of the new version of Atomer was able to derive suitable contracts *automatically*. Subsequently, Atomer found no atomicity violations in such corrected programs.

[2] **Experiments on validation programs** derived from **Gluon**: `https://github.com/harmim/vut-pp1/tree/master/testing-programs/contracts-first-experiments-atomer`.

## 6.3 Experiments with Real-Life Programs

Since the new version of Atomer supports the analysis of Java programs, two *open-source real-life extensive* (both ~250,000 lines of code) Java programs were analysed — *Apache Cassandra*[3] 3.11 and *Apache Tomcat*[4] 8.5. In [13], there were reported several *atomicity-related bugs* in these programs. It turned out that the reported bugs were real atomicity violation errors, and they were later fixed.

When Atomer first analysed these programs (at that time, without most of the extensions presented in this thesis), the bugs were successfully *rediscovered*, but quite some *false alarms* were reported. However, after all the improvements proposed in the thesis were implemented, the number of false alarms was *significantly reduced*. In particular, the support for *re-entrant locks* increased the analysis' precision a lot because these types of locks appear pretty often in these programs. Moreover, many "large" atomic sections in these programs dramatically increase the number of reported false alarms. Therefore, the *parameters of the analysis* presented in the thesis were used. In particular, the maximum length of critical sections was limited with the parameter $d$ set to 20, and the number of levels considered during analysing nested functions was limited with the parameter $r$ set to 10. Further, also using the parametrisation of the analysis, several "non-critical" functions were ignored during the analysis (e.g., `String.format`, `*.toString`, `*.toArray`, `Log.debug`, `Integer.valueOf`, etc.).

With all these Atomer's extensions, when analysing one of the source files of Tomcat where a real atomicity violation was reported before (i.e., `org.apache.catalina.core.StandardContext` — this package itself contains 6,684 lines of code, and the packages it depends on, that must have been analysed too, contain tens of thousands lines of code), the number of reported errors decreased from ~800 to 228. Then, when analysing one of the source files of Cassandra where a real atomicity violation was reported before (i.e., `org.apache.cassandra.streaming.StreamSession` — this package itself contains 1,014 lines of code, and the packages it depends on, that must have been analysed too, contain tens of thousands lines of code), the number of reported errors decreased from ~700 to 112.

Obviously, most of the previously reported errors were false alarms. However, it is still challenging to say which of these errors are real atomicity violations. Indeed, the author suspects some of the warnings correspond to real errors, but so far, the author has not managed to confirm that.

## 6.4 Summary of the Evaluation and Future Work

Using small *hand-crafted programs*, it was *successfully validated* that the implemented new version of Atomer works correctly w.r.t. the proposal. This validation includes all *scalability/precision* enhancements as well as support for other programming languages and *advanced locking mechanisms* that were not supported in the first version of Atomer.

[3] Open-source NoSQL database system **Apache Cassandra**: `https://cassandra.apache.org`.
[4] Open-source Java HTTP web server **Apache Tomcat**: `https://tomcat.apache.org`.

*Scalability testing* in Section 6.1 shows that the new version of Atomer can analyse quite *extensive real-life low-level* programs in a reasonable time. Moreover, it was proven that the new version of Atomer is *significantly faster* than the first version implemented in [17]. This improvement is caused by the *approximation* of working with *sequences of function calls* by working with *sets of function calls*. Regarding performance, Atomer may be directly usable for large real-life projects. However, the performance of Atomer strongly depends on the size of *critical sections* (i.e., the number of function calls inside critical sections). Thus, it is needed to filter out "large" critical sections using the proposed parameters in some cases.

Sections 6.2 and 6.3 demonstrate that the new version of Atomer can also reveal *real atomicity-related bugs* in C++ and Java programs. In the case of the complicated and large Java programs from Section 6.3, it was shown that the accuracy of the new version of Atomer was *significantly improved*. However, unfortunately, quite some false alarms are still reported along with real errors. It can be demanding to properly classify which of the reported errors are real errors and false alarms. This can make the analysis unusable on some kinds of real-world programs. But still, it is possible to find appropriate values of the proposed parameters for a particular program to obtain good results. For instance, in this way, one can specify several specific "critical functions" that Atomer should focus on. This should often lead to much better results.

The above implies that future work will focus mainly on further improvements of the accuracy of the analysis. Besides, the analysis results may be used as an input for *dynamic analysis*, checking whether the atomicity violations are real errors. For example, one could use they ANaConDA [14] dynamic analyser, which uses *noise-based testing* with *extrapolated checking* for violations of contracts for concurrency. ANaConDA could be instructed to concentrate its analysis and *noise injection* to those sets whose atomicity was found broken. Furthermore, it seems promising to consider *formal parameters* and *distinguish the context* of called functions during the analysis to reduce false alarms.

Another exciting idea is to use *machine learning* to learn appropriate values of the analysis' parameters (introduced in this thesis) for particular programs. *Automation* of the analysis can be preserved using this approach, and the precision of the analysis may be high enough (however, the precision would highly depend on *training data*). Another solution to reducing the number of false alarms is to do some *statistical ranking* of reported errors and display only the most critical errors to the user.

During experimenting with Tomcat, the author of this thesis found out that several reported errors were due to calling functions with parameters that were *local* variables (i.e., *non-shared* variables). These cases cannot lead to real atomicity violations. Therefore, it would be helpful to statically identify whether function parameters are shared variables. If they are not, calls of such functions should be ignored.

# Chapter 7

# Conclusion

The thesis started by describing the principles of program analysis with a focus on *static analysis* and *abstract interpretation*. Further, *Facebook Infer* was described — a concrete static analysis framework that uses abstract interpretation. Next, there were described *contracts for concurrency*. The major part of the thesis then aimed to describe static analyser *Atomer*[1] — proposed and implemented within the author's bachelor's thesis [17] — implemented as a Facebook Infer's module for detection of *atomicity violations* in *multi-threaded programs*. It was described its limitations, and thereafter, it was introduced the proposal and implementation of its extensions and improvements. Lastly, the experimental evaluation of the new features and improvements was discussed together with possible future work.

The first version of Atomer works on the level of *sequences of function calls*. It is based on the assumption that sequences of function calls executed *atomically once* should probably be executed *always atomically*, and it naturally works with sequences. In the thesis, to improve *scalability*, the use of sequences was *approximated* by *sets*. Furthermore, several new features were implemented, e.g., support for *C++* and *Java*, including various *advanced kinds of locks* these languages offer (such as *re-entrant locks* or *lock guards*); or a more precise way of *distinguishing between different lock instances*. Moreover, the analysis has been *parametrised*, so the user can provide some input values to the analysis to make the analysis more accurate.

The introduced enhancements were *successfully tested* and *experimentally evaluated* also on *extensive real-life software* where *real bugs* were *successfully rediscovered*. It turned out that such innovations improved the *accuracy* and *scalability* of the analysis a lot. However, Atomer's accuracy can be further increased. There are still some other improvements and ideas to work on, for instance, considering *formal parameters* and *distinguishing the context* of called functions; or combinations with a *dynamic analysis*. Another interesting idea is to use *machine learning* to learn appropriate values of the analysis' parameters for particular programs. Further, it is needed to perform more experiments on real-life programs to find and report *new bugs*.

It is expected that the work on this project will continue within the VeriFIT group at FIT BUT. The preliminary results of the thesis were presented at the Excel@FIT'21 conference, where it *won two awards*.

[1] The implementation of a new version of **Atomer** is available at GitHub as an *open-source* repository: `https://github.com/harmim/infer/tree/atomer-v2.0.0`.