


Ultimate GemCutter and the Axes of Generalization
(Competition Contribution)

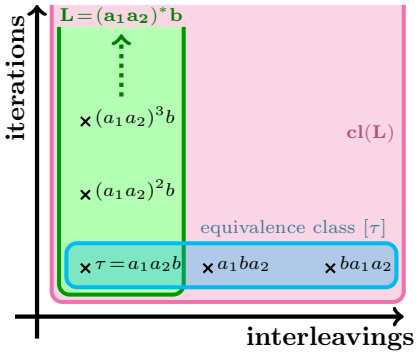
Dominik Klumpp*[1]✉, Daniel Dietsch[1], Matthias Heizmann[1], Frank Schüssele[1], Marcel Ebbinghaus[1], Azadeh Farzan[2], and Andreas Podelski[1]

[1] University of Freiburg, Freiburg im Breisgau, Germany
klumpp@informatik.uni-freiburg.de
[2] University of Toronto, Toronto, Canada

Abstract. Ultimate GemCutter verifies concurrent programs using the CEGAR paradigm, by generalizing from spurious counterexample traces to larger sets of correct traces. We integrate classical CEGAR generalization with orthogonal generalization across interleavings. Thereby, we are able to prove correctness of programs otherwise out-of-reach for interpolation-based verification. The competition results show significant advantages over other concurrency approaches in the Ultimate family.

1 Verification Approach

Ultimate GemCutter is a verification tool for concurrent programs based on the CEGAR paradigm: It *(1)* picks a trace from the set of all program interleavings (a possible *"counterexample"*), *(2)* proves correctness of this trace (the counterexample is *"spurious"*), and *(3)* generalizes the proof to conclude that a larger (usually infinite) set of traces is correct. Classically, CEGAR focuses on generalization across traces with varying numbers of loop iterations, by finding inductive loop invariants. GemCutter proposes additional generalization along an orthogonal axis: *across interleavings*.



Concurrent programs contain many redundant interleavings of actions from different threads, i.e., interleavings with the same (input/output-) behaviour. A naïve application of CEGAR requires explicit proofs of correctness for all these interleavings. Intermediate states during execution of redundant interleavings differ, and different interleavings often require different correctness proofs. GemCutter addresses this as illustrated in the figure on the right: We prove correctness of a trace τ, here $\tau = a_1a_2b$, where a_1, a_2 are actions of the first thread,

* Jury Member: Dominik Klumpp

D. Fisman and G. Rosu (Eds.): TACAS 2022, LNCS 13244, pp. 479–483, 2022.
https://doi.org/10.1007/978-3-030-99527-0_35

and b is an action of the second thread. The proof of correctness is generated using Craig interpolation or similar techniques. We generalize this proof into a Floyd-Hoare automaton [8] to show that a regular language L (green area in the figure above) of traces is correct. The new contribution is the subsequent generalization step: If a trace τ_1 differs from a (correct) trace τ_2 in L only by the ordering of *independent* statements, these traces are *(Mazurkiewicz-) equivalent* [3]. We conclude that τ_1 is also correct. Hence, the set of all such traces, denoted $cl(L)$ (pink area), contains only correct traces. If the set of all program interleavings P is a subset of $cl(L)$, we conclude that the program is correct.

To soundly make this conclusion, we need a suitable notion of *independence* between statements, which guarantees that the order of execution of two independent statements does not matter for program correctness. An intuitive sufficient condition is that neither statement writes to a memory location read or written by the other statement. If we cannot establish this condition syntactically, we use an SMT solver to check if executing the statements in either order is guaranteed to give the same result. We use information from the Floyd-Hoare automaton to refine this check in the style of *conditional independence* [5]. Such information can for instance express (but is not limited to) non-aliasing of pointers.

However, the inclusion $P \subseteq cl(L)$ is in general undecidable [3]; $cl(L)$ may not be regular. We reverse our viewpoint to provide a sufficient condition that can be effectively checked: Rather than adding all equivalent traces to L – thus obtaining $cl(L)$ –, we instead remove all but one trace of each equivalence class from P – yielding a *reduction* P' of P (formally, $cl(P') = P$). We use the *sleep set* technique [5] to remove transitions from an automaton for P to get an automaton that recognizes one such reduction P'. We then check whether the (regular) reduction P' is included in the (regular) language L. If this inclusion $P' \subseteq L$ holds, it implies that $P \subseteq cl(L)$ also holds, and the program is correct. If the inclusion does not (yet) hold, GemCutter picks another program trace and repeats the process, iteratively building up the language L of correct traces by taking the union of the Floyd-Hoare automata computed in all iterations.

A key feature of the reduction-based approach is that the generalization along the iteration and interleaving axes is combined not just additively, but multiplicatively: In the geometrical intuition of the figure above, we do not just take the union of L (green area) with the equivalence class $[\tau]$ of τ (blue area), but consider all traces in $cl(L)$ (the pink area which is spanned by both). Further, we heuristically try to pick a set of representatives in a way that harmonizes with CEGAR generalization, i.e., a reduction P' with simple loop invariants. To this end, we prefer representatives with context-switches at all loop boundaries. Ideally, each thread performs one complete loop iteration and then hands control over to the next thread (the last thread hands back control to the first thread). Consider the example program on the right, with the postcondition $x = y$. Here, a proof for the

```
// Thread 1:
int x = 0;
for (int i = 0; i < N; ++i) {
  x += A[i];
}
```

```
// Thread 2:
int y = 0;
for (int j = 0; j < N; ++j) {
  y += A[j];
}
```

set of all interleavings P, or some inopportunely chosen reduction, needs invariants that capture the fact that $x = \sum_{k=0}^{i} A[k]$, and similar for y. Such invariants are usually not found by Craig interpolation. However, the loop invariant $i = j \wedge x = y$ suffices for the reduction that places context-switches at all loop boundaries. The general idea is that for this kind of reduction, the proof often needs to summarize only the effect of a single loop iteration rather than unboundedly many iterations (which may require quantifiers or non-linear arithmetic). Similar observations were first made by Farzan and Vandikas [4].

GemCutter furthermore aims to improve efficiency of the proof check, i.e., the check whether a reduction P' is a subset of the set of proven traces L. The state explosion problem of concurrent programs makes the computation of an automaton recognizing a reduction P' as well as the subsequent inclusion check prohibitively expensive. To address this, we implemented a form of *persistent set reduction* [5], which allows us to compute a more compact automaton recognizing P'. This results in a more time- and memory-efficient inclusion check.

Reductions that interact harmoniously with CEGAR generalization do not always allow for an efficient proof check, nor vice versa. In the ConcurrencySafety category, where correctness proofs may become complicated, we prioritize generalization by computing reductions that typically allow for simpler proofs (described above), even though proof checking for such reductions is often more expensive. By contrast, in the NoDataRace category we found proof assertions to be usually quite simple (often only expressing non-aliasing of pointers), so we prioritize faster proof checks (and postpone context-switches as far as possible).

Implementation GemCutter uses the libraries and the front-end of the Ultimate framework, and extends Ultimate with a new CEGAR loop implementation and new algorithms operating on finite automata. We represent programs P, reductions P' and sets of proven traces L as finite automata. Ultimate constructs Floyd-Hoare automata (for L) only on-demand [7]. Due to the state explosion problem, GemCutter extends this approach to the program and the reduction. The necessary parts of the automata are constructed just-in-time during traversal by automata algorithms. Various techniques are implemented as instances of a few generic interfaces (on-demand automata, and visitors that monitor and guide automaton traversal) for flexibility: Radically different algorithms can be created by configuring, exchanging and stacking interface implementations. The following techniques and optimizations (all used in SV-COMP) can be combined with each other independently: **(i)** sleep set reduction; **(ii)** persistent set reduction; **(iii)** discovery and pruning of states that cannot reach accepting states; **(iv)** guidance towards representatives of a specific form, e.g. with context-switches at loop boundaries; and **(v)** inclusion check between automata.

2 Strengths and Weaknesses

The main advantage over other concurrency approaches in Ultimate (in Automizer and Taipan) lies in the generalization across interleavings: Automizer

and Taipan typically require more complex proofs possibly out-of-reach for Craig interpolation and similar techniques. GemCutter performs significantly better, winning 3rd place in the ConcurrencySafety category (behind the bounded model checkers Deagle [6] and CSeq [10]) and 1st place in the NoDataRace demo category. For details, refer to the competition report [1].

Since our proof check decides a stronger condition ($P' \subseteq L$), it might miss some cases in which the proof is actually sufficient, i.e., $P \subseteq cl(L)$ holds. This is because P' and L might contain different representatives for the same equivalence class of interleavings. This weakness cannot be resolved completely due to the undecidability of the inclusion $P \subseteq cl(L)$. It can however be attenuated by considering other choices of representatives (other than preferring context-switches at loop boundaries) and exploring the effect. This choice is currently given as an input parameter; an approach that heuristically chooses a reduction based on the program structure might perform better. Our notion of independence between statements is currently ignorant of the specification being verified. We hope to extend our approach to take this into account. Finally, our approach (and implementation) can be easily extended with other reduction methods that correspond to more aggressive generalization along the interleaving axis.

Our approach only verifies programs with a bounded number of threads. GemCutter runs out of time or memory if it is unable to establish such an upper bound, e.g. for many benchmarks in `pthread-ext/` or `goblint-regression/`.

3 Architecture, Setup, Configuration, and Project

GemCutter is part of the program analysis framework Ultimate[3], written in Java and licensed under LGPLv3[4]. GemCutter version 0.2.2-839c364b requires Java 11 and Python 3.6. Its Linux version, binaries of the required SMT solvers[5], and a Python wrapper script were submitted as a `.zip` archive. GemCutter is invoked with

```
./Ultimate.py --spec <p> --file <f> --architecture <a> --full-output
```

where `<p>` is an SV-COMP property file, `<f>` is an input C file, `<a>` is the architecture (`32bit` or `64bit`), and `--full-output` enables verbose output to `stdout`. A violation witness may be written to the file `witness.graphml`. The benchmarking tool BenchExec [2] supports GemCutter through the tool-info module `ultimategemcutter.py`[6]. GemCutter participates in the ConcurrencySafety and NoDataRace categories, as declared in its SV-COMP benchmark definition file `ugemcutter.xml`[7].

Data Availability Our `.zip` archive is available online[8] and on Zenodo [9].

[3] ultimate.informatik.uni-freiburg.de and github.com/ultimate-pa/ultimate

[4] www.gnu.org/licenses/lgpl-3.0.en.html

[5] Z3 (github.com/Z3Prover/z3), CVC4 (cvc4.github.io) and Mathsat (mathsat.fbk.eu)

[6] github.com/sosy-lab/benchexec/blob/main/benchexec/tools/ultimategemcutter.py

[7] gitlab.com/sosy-lab/sv-comp/bench-defs/-/blob/main/benchmark-defs/ugemcutter.xml

[8] gitlab.com/sosy-lab/sv-comp/archives-2022/-/blob/main/2022/ugemcutter.zip and git.io/JM69B

References

1. Beyer, D.: Progress on software verification: SV-COMP 2022. In: Proc. TACAS (2). Springer (2022)
2. Beyer, D., Löwe, S., Wendler, P.: Reliable benchmarking: requirements and solutions. Int. J. Softw. Tools Technol. Transf. **21**(1), 1–29 (2019). https://doi.org/10.1007/s10009-017-0469-y
3. Diekert, V., Rozenberg, G. (eds.): The Book of Traces. World Scientific (1995). https://doi.org/10.1142/2563
4. Farzan, A., Vandikas, A.: Automated hypersafety verification. In: CAV (1). Lecture Notes in Computer Science, vol. 11561, pp. 200–218. Springer (2019). https://doi.org/10.1007/978-3-030-25540-4_11
5. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem, Lecture Notes in Computer Science, vol. 1032. Springer (1996). https://doi.org/10.1007/3-540-60761-7
6. He, F., Sun, Z., Fan, H.: Deagle: An SMT-based verifier for multi-threaded programs (competition contribution). In: Proc. TACAS (2). Springer (2022)
7. Heizmann, M., Chen, Y., Dietsch, D., Greitschus, M., Nutz, A., Musa, B., Schätzle, C., Schilling, C., Schüssele, F., Podelski, A.: Ultimate Automizer with an on-demand construction of Floyd-Hoare automata - (competition contribution). In: TACAS (2). Lecture Notes in Computer Science, vol. 10206, pp. 394–398 (2017). https://doi.org/10.1007/978-3-662-54580-5_30
8. Heizmann, M., Hoenicke, J., Podelski, A.: Refinement of trace abstraction. In: SAS. Lecture Notes in Computer Science, vol. 5673, pp. 69–85. Springer (2009). https://doi.org/10.1007/978-3-642-03237-0_7
9. Klumpp, D., Dietsch, D., Heizmann, M., Schüssele, F., Ebbinghaus, M., Farzan, A., Podelski, A.: Ultimate GemCutter SV-COMP 2022 Competition Contribution (Nov 2021). https://doi.org/10.5281/zenodo.5956945
10. Sales, E., Coto, A., Inverso, O., Tuosto, E.: A prototype for data race detection in CSeq 3 (competition contribution). In: Proc. TACAS (2). Springer (2022)