A Prototype for Data Race Detection in CSeq 3[*]
(Competition Contribution)

Alex Coto, Omar Inverso, Emerson Sales✉, and Emilio Tuosto

Gran Sasso Science Institute, L'Aquila, Italy
{alex.coto,omar.inverso,emerson.sales,emilio.tuosto}@gssi.it

Abstract. We sketch a sequentialization-based technique for bounded detection of data races under sequential consistency, and summarise the major improvements to our verification framework over the last years.

Keywords: Bounded model checking · Context-bounded analysis · Sequentialization · Data races · Reachability · Concurrency · Threads

1 Verification Approach

Our approach is based on *lazy sequentialization* [7]. The idea is to convert the concurrent program P of interest into a non-deterministic sequential program $Q_{u,k}$ that preserves all feasible executions of P up to unwinding bound u and k rounds (or execution contexts [8]). Among different techniques [6], we choose *bounded model checking* [3] to analyse $Q_{u,k}$. In this section, we briefly overview lazy sequentialisation, and sketch a novel extension to detect data races. Further elements of novelty w.r.t. engineering of our tool are discussed in the next section.

Lazy Sequentialization. We unwind all loops and inline all functions in P, except the main function and those from which a thread is spawned, obtaining a *bounded program* P_u that preserves all feasible executions of P up to the unwinding bound u. We then transform each function of P_u into a *thread simulation function* where each *visible statement* is assigned a numerical label and a guard, and each call to a concurrency-specific function is replaced by a call to a function that models the same intended semantics; for each simulation function, we add a global variable to represent the *program counter*, initially set to zero.

A thread's execution context of P_u is simulated by invoking the corresponding thread simulation function of $Q_{u,k}$ that executes from the first statement to a non-deterministically selected label, updates the program counter, and returns. Further execution contexts are simulated by re-invoking the simulation function, where the guards ensure that the control is repositioned to the correct numerical label via a sequence of jumps, and so on. To retain consistency of the local state of the thread across different invocations of the simulation functions, static storage

[*] This work has been partially funded by MIUR project PRIN 2017FTXR7S IT-MATTERS and MUR project FISR2020IP_05310 MVM-Adapt.

is enforced for all local variables. We drive the overall simulation of P_u from the main function of $Q_{u,k}$, by invoking the thread simulation functions appropriately.

Data Race Detection. A program contains a *data race* if it can execute two *conflicting actions* (i.e., one thread modifies a memory location and another one reads or modifies the same location), at least one of which is not atomic, and neither happens before the other [9]. Consider two threads performing the operation `v = v + 1` on a shared variable initialised to zero. Both threads try to modify the data at the memory location reserved for `v`, but the necessary sequences of memory accesses are not synchronised, and thus may interleave. If a context-switch happens between the memory read and write operations in the thread that runs first, both threads will read 0, and at the end of the execution the value of `v` will be 1. To detect such situation, we alter the encoding from P_u to $Q_{u,k}$ by (i) adding a shared array `w_addrs` that stores a pointer to the memory location targeted by a write operation for each thread, (ii) injecting additional control code at each visible statement, and (iii) splitting the modified sequentialised encoding of the visible statement into two separate sequentialised statements to allow in-between context switching. The code fragment shows the modified sequentialised encoding (no guards for simplicity, injected code greyed out) for the statement `v = v + 1` of the first thread of the program described above. We store in `w_addr` the address of the variable being written, and then assert that the other thread is not writing to the same location; in the same (simulated) execution context, we store `w_addr` in `w_addrs`, so that the assertion can be checked within the other thread too. We reset `w_addrs` right after the statement under consideration. Note the label `k+1` that allows thread pre-emption. Now, one of the threads can execute the simulated statement at label `k` and context-switch at label `k+1` while `w_addrs` still points to `v`; this makes it possible to schedule the other thread, and fail the assertion in there.

```
k:
 void *w_addr = &v;
 assert(w_addrs[1] != w_addr);
 w_addrs[0] = w_addr;
 v = v + 1;
k+1:
 w_addrs[0] = 0;
```

In the general case, handling multiple memory write accesses for a single statement requires a slightly different tracking mechanism for write addresses, or decomposition into simpler statements. Statements with read-only shared memory access are handled without updating `w_addrs`. Programs with more than two threads require multiple assertions.

2 Software Architecture

CSeq is a framework for quick development of static analysis and program transformation prototypes. For parsing the input program CSeq relies on pycparserext (`pypi.org/project/pycparserext`), an extension of pycparser (`github.com/eliben/pycparser`), which in turn is built on top of PLY (`www.dabeaz.com/ply`), a Python implementation of Lex and Yacc. All the mentioned components as well as CSeq are entirely written in Python.

We combined several groups of modules in CSeq, namely (i) program simplification, (ii) program unfolding, (iii) sequentialization, (iv) instrumentation, and (v) backend invocation and counterexample generation. For the analysis of the sequentialised program we rely on CBMC (`www.cprover.org/cbmc`), that in turn embeds the DPLL-style MiniSat SAT solver (`minisat.se`).

CSeq 3.0 incorporates a significant number of enhancements. At an architectural level, the main element of novelty is in the modularity between the general-purpose functionalities of the framework and the specific lazy sequentialization, which opens up to the possibility of prototyping different static analysers for other applications (e.g., [11,10]) as well as improving older sequentialization-based prototypes (e.g., [4,12,13] and variations thereof). The enhancements to the framework include: Python 3 support, support for GNU C compiler extensions, a fully re-implemented symbol table, revised general-purpose modules such as constant propagation, function inlining, and loop unrolling, and a custom-built version of CBMC (not used in the competition) for SAT-solving under assumptions. For the competition we include (experimental) enhanced constant propagation, and simplified function inlining. Besides the data race checking extension, the sequentialization modules include improvements from earlier implementations [5,8,6] and for different editions of SV-COMP up to date, in particular: extended pthread API support (conditional waiting, barriers, and thread-specific data management), context-bounded analysis, and a major code overhaul.

3 Strengths and Weaknesses

The table below summarises the performance of our tool on the 764 cases of the `Concurrency` category and the 162 cases of the data race demo category.

Overall instances		764	162
Correct	safe	202	37
	unsafe	320	61
Unknown	reject	9	19
	internal error	18	17
	out of time	159	20
	out of memory	56	2
Incorrect	safe	0	0
	unsafe	0	6

Our technique excels at hunting bugs, as shown by the number of correct unsafe (incl. 17 malformed witnesses and 50 unconfirmed witnesses), but gets quickly expensive with larger bounds, hitting the resource limits. The additional context-switch points and the use of pointers for data race detection introduce further overhead. The other failures are due to limiting assumptions or glitches in the implementation. All the false positives are due to corner cases in the encoding.

4 Setup and Configuration

We competed in the `ConcurrencySafety` category and in the data race detection demo category. CSeq 3.0 is available at `https://github.com/omainv/cseq/releases`.

Installation instructions are in the `README` file within the package. A wrapper script (`lazy-cseq.py`) invokes CSeq up to three times, with the options `-l lazy` for lazy sequentialisation, `--sv-comp` to enable the required violation witnesses format, `--atomic-parameters` to assume atomic passing of function arguments, `--nondet-condvar-wakeups` for non-deterministic spurious conditional variables wake-up calls, `--deep-propagation` for experimental constant folding and propagation, `--32` for 32-bit architectures, `--threads 100` to limit the overall number of threads, `--data-race-check` when required, and `--backend cbmc` to use CBMC 5.4 for sequential analysis.

For reachability checking, on different invocations the script adds different parameters: `-r2 -w2 -f2`, `-r4 -w3 -f5`, and `-r20 -w1 -f11`, where `r` is the number of rounds, and `f` and `w` are the unwind bounds for `for` (i.e., potentially bounded) and `while` (i.e., potentially unbounded) loops, respectively; on the last invocation `--softunwindbound` and `--unwind-for-max 10000` are also added to fully unfold `for` loops if a static bound can be found, up to the given hard bound. For data race detection, the above parameters are replaced with `-c4 -u2`, `-c10 -u10`, and `-c50 -w20 -f20` with `--unwind-for-max 100`. Note that in this case the bound is on the number of execution contexts rather than rounds (`-c` vs. `-r`), and `-u` is used as a shorthand for `-f` and `-w`.

We leave the analysis running to completion every time. When the result is `TRUE`, the scripts restarts the analysis with the next set of parameters. As soon as the script gets `FALSE`, it returns `FALSE`. Only if the analysis using the last set of parameters is finished and the result is `TRUE`, then the script returns `TRUE`.

Data Availability Statement. All data of SV-COMP 2022 are archived as described in the competition report [1] and available on the competition web site. This includes the verification tasks, results, witnesses, scripts, and instructions for reproduction. The version of our verifier as used in the competition is archived together with other participating tools [2].

References

1. Beyer, D.: Progress on software verification: SV-COMP 2022. In: Proc. TACAS. Springer (2022)
2. Beyer, D.: Verifiers and validators of the 11th Intl. Competition on Software Verification (SV-COMP 2022). Zenodo (2022). https://doi.org/10.5281/zenodo.5959149
3. Clarke, E.M., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: TACAS. Lecture Notes in Computer Science, vol. 2988, pp. 168–176. Springer (2004). https://doi.org/10.1007/978-3-540-24730-2_15
4. Fischer, B., Inverso, O., Parlato, G.: Cseq: A concurrency pre-processor for sequential C verification tools. In: ASE. pp. 710–713. IEEE (2013). https://doi.org/10.1109/ASE.2013.6693139
5. Inverso, O., Nguyen, T.L., Fischer, B., Torre, S.L., Parlato, G.: Lazy-cseq: A context-bounded model checking tool for multi-threaded c-programs. In: ASE. pp. 807–812. IEEE Computer Society (2015). https://doi.org/10.1109/ASE.2015.108
6. Inverso, O., Tomasco, E., Fischer, B., La Torre, S., Parlato, G.: Bounded verification of multi-threaded programs via lazy sequentialization. ACM Trans. Program. Lang. Syst. **44**(1) (dec 2021). https://doi.org/10.1145/3478536
7. Inverso, O., Tomasco, E., Fischer, B., Torre, S.L., Parlato, G.: Bounded model checking of multi-threaded C programs via lazy sequentialization. In: CAV. Lecture Notes in Computer Science, vol. 8559, pp. 585–602. Springer (2014). https://doi.org/10.1007/978-3-319-08867-9_39

8. Inverso, O., Trubiani, C.: Parallel and distributed bounded model checking of multi-threaded programs. In: PPoPP. pp. 202–216. ACM (2020). https://doi.org/10.1145/3332466.3374529
9. ISO/IEC: ISO/IEC 9899:2018: Information technology – Programming languages – C (Jun 2018)
10. Simic, S., Bemporad, A., Inverso, O., Tribastone, M.: Tight error analysis in fixed-point arithmetic. In: IFM. Lecture Notes in Computer Science, vol. 12546, pp. 318–336. Springer (2020). https://doi.org/10.1007/978-3-030-63461-$2_1$7
11. Simic, S., Inverso, O., Tribastone, M.: Bit-precise verification of discontinuity errors under fixed-point arithmetic. In: SEFM. Lecture Notes in Computer Science, vol. 13085, pp. 443–460. Springer (2021). https://doi.org/10.1007/978-3-030-92124-8_25
12. Tomasco, E., Inverso, O., Fischer, B., Torre, S.L., Parlato, G.: Verifying concurrent programs by memory unwinding. In: TACAS. Lecture Notes in Computer Science, vol. 9035, pp. 551–565. Springer (2015). https://doi.org/10.1007/978-3-662-46681-0_52
13. Tomasco, E., Nguyen, T.L., Inverso, O., Fischer, B., Torre, S.L., Parlato, G.: Lazy sequentialization for TSO and PSO via shared memory abstractions. In: FMCAD. pp. 193–200. IEEE (2016). https://doi.org/10.1109/FMCAD.2016.7886679