Korn—Software Verification with Horn Clauses (Competition Contribution)

Gidon Ernst(✉)*

LMU Munich, Munich, Germany
gidon.ernst@lmu.de

Abstract. Korn is a software verifier that infers correctness certificates and violation witnesses sutomatically using state-of-the-art Horn-clause solvers, such as Z3 and Eldarica. The solvers are used in a portfolio together with cheap random sampling where the latter can be very effective at finding counterexamples. Korn perfomend best in the `Recursive` sub-category of SV-COMP 2023.

Keywords: Software Verification · Horn Clauses · Loop Contracts

1 Verification Approach

Korn is a verifier for C programs that is based on a translation into systems of constrained Horn clauses [5,12]. Therein, each program location is abstracted by a second-order predicate over the program variables which are active at that point. The system of Horn clauses has a (second-order) solution if and only if the program is correct. Horn clauses encodings are a convenient intermediate representation that is linear in the size of the program and that is inherently modular, such that loops, procedure contracts, and non-local control flow like `goto`s and labels can be easily abstracted (see Sect. 3 wrt. category `Recursive`).

Korn uses state-of-the-art solvers to determine the satisfiability of the generated Horn clause system (cf. Sect. 2), specifically for SV-COMP it uses Z3 [6] and Eldarica [15]. Both solvers generate evidence for correctness of a given program in terms of models that describe how the unknown predicates need to be instantiated. Moreover, Eldarica can generate counterexample traces, and Korn instruments the Horn clause system to get the concrete values returned by the `__VERIFIER_nondet_*()` functions on an error path. For these reasons, Korn tends to produce detailed correctness and violation witnesses.

The different solvers have different strengths and weaknesses. To that end, Korn implements a portfolio approach with several sequential stages. The configuration for SV-COMP 2023 [2] is as follows, where the specific timeouts for the individual tools are chosen heuristically based on prior experiments:

1. Initially, 10s of random sampling with small values is performed. It picks for each input value uniformly between number 0, and values of 2, 5, and 10 bits respectively, possibly with a sign. Absense of too large values avoids

* Jury Member

S. Sankaranarayanan and N. Sharygina (Eds.): TACAS 2023, LNCS 13994, pp. 559–564, 2023.
https://doi.org/10.1007/978-3-031-30820-8_36

very long running loops when the counter is nondeterministic. There is no particular justification for the sampling scheme, but it is effective.
2. Next, Z3 is executed on the verification problem, translated from C to Horn clauses for 20s. Usually, Z3 finds solutions very quickly if it succeeds at all, specifically on those benchmarks where Z3 succeeds but not Eldarica.
3. Finally, Eldarica is executed for the remaining time. From past experience, it should be slightly better in comparison to Z3 in the long run on this specific set of tasks [10]. The generated invariants from Eldarica tend to be simpler and avoid the existential quantifiers often introduced by Z3, which improves witness generation. To prevent spurious counterexamples, Korn reports a violations only if it can be confirmed by executing the program natively.

Korn is overall similar to SeaHorn [13] but it operates on the C source level instead of LLVM. Korn aims at a rather different design point, namely to favor simplicity over features, therefore offering a good platform for experiments. Eldarica has its own C frontend that supports a different set of features, recently published as TriCera [11]. Here the main distinction is that Korn uses a large block encoding, such that the verification conditions closely reflect the structure of the program. Korn offers a second verification approach with loop contracts [16,14,7]. This was the original motivation to develop the tool, and neither SeaHorn nor TriCera supports this feature, albeit it was not used for SV-COMP because it offers no advantages [10] and because the encoding of loop contracts into loop invariants would require quantifiers in the witnesses format.

2 Software Architecture

Korn is mainly written in the JVM language Scala.[1] The front-end uses a custom parser, generated with jFlex and Beaver. The random sampler relies on native execution which links the benchmark task with a C file `__VERIFIER_random.c` that implements the `_VERIFIER_nondet_*` functions. Verification conditions are generated in the fragment of SMT-LIB of the `HORN` logic.[2] Korn can invoke any compliant solver as a backend either using its standard input or a file to communicate the verification task. There is explicit support for Z3 [12], Eldarica [15] to pass e.g. timeouts with tool-specific options or to produce models resp. counterexamples. Currently, Korn use the theories of integers and arrays.

In order to produce SV-COMP correctness witnesses, Korn can read the models generated by the backend-solvers, and translate them back into C expressions. The correctness witnesses produced currently are derived from the invariants that are reported back by the Horn solvers (`get-model` resp. `-ssol` flag of Eldarica). Violation witnesses are either read off the output of Eldarica (`-cex` flag), or from the output of the random sampler, as a sequence of nondeterministic choices. When a counterexample is found, a test harness is compiled to confirm whether `reach_error()` is in fact called.

[1] https://scala-lang.org
[2] https://chc-comp.github.io/format.html

3 Discussion: Strengths and Weaknesses

Korn supports a substantial fraction of the C language, with the greatest limitation being the lack of support for dynamic data structures (see website for a detailed account), which means that currently any task which requires a memory model is out of scope. The translation supports most control structures, including `goto` and labels. With respect to solving verification tasks, Korn inherits the strenths and limitations of the underlying solvers. Tasks that for which invariants and procedure contracts are expressible in linear integer arithmetic are typically proved quickly by the solvers, whereas they struggle on tasks with arrays and quantified invariants. Honoring these aspects, Korn participated in four categories, `ControlFlow`, `Loops`, `Recursive`, `XCSP` for property `ReachSafety`.

The theoretical approach used by Korn is sound and complete relative to the solver capabilities. Korn produced no incorrect result in SV-COMP 2023, but there are circumstances which could lead to wrong verdicts. With respect to C semantics, Korn currently makes the following trade-offs:

- Integer types are treated as unbounded and arithmetic overflows are not modeled at all. This affects a single task, `nla-digbench/geo1-u.c`, which contains an error caused by an unsigned integer overflow. This error is fortunately caught by random sampling— Korn would otherwise wrongly prove this task safe. We aim to experiment with a bitvector encoding eventually, which would allow Korn to tackle tasks involving bitwise operations.
- Arrays are currently modeled as value types. Benchmarks in which tracking aliases is relevant may not be solved correctly, but that does not occur in the categories in which Korn participates.
- By confirming counterexamples via native execution, each bug reported is necessarily a true bug. This safety net catches two incorrect error verdicts on `loops-crafted/theatreSquare.c` and `recursive/Primes.c`, the reason for this unsoundness is under investigation. However, counterexample confirmation prevents Korn from rightfully reporting 50 error verdicts found by Z3 in category `XCSP` which are missed by Eldarica († in Sect. 1). It is unclear how to get usable counterexample traces from Z3 to resolve this dilemma.
- Differently from most other SV-COMP tools, Korn fixes the evaluation order of function arguments to be right-to-left which matches the order typically used by C compilers. This is not faithful to C semantics as Korn potentially misses bugs due to side-effects for some specific evaluation order.

The random sampler is very effective—in SV-COMP 2023 it discovered all 210 violations reported by Korn, of which 204 are found within 2 seconds. Sampling of small *non-zero* values is crucial, e.g., `Ackermann02.c` falsifies with input vector `[2,0]`; using all zero inputs still finds 57 of these 210 violations.

A key strength of Horn clause encodings is that they are inherently modular. This means that loops and recursion are abstracted by invariants resp. pre-/postcondition pairs. The latter enable Korn to significantly outperform all other tools in category `Recursive`. Plausible explanations are that classic state-space exploration techniques struggle to abstract call stacks or maybe that

Table 1. Comparison of official results (number of tasks solved) in comparison to result of the best-scoring other tool in that category and post-competition experiments after fixing an issue with the submitted Korn verifier archive which did not run Eldarica at all. # Tasks is the number of tasks supported by Korn vs. category size. The result marked by † is without counterexample confirmation. The official results can be found at https://sv-comp.sosy-lab.org/2023/results/results-verified/

		SV-COMP 2023					Post-Comp.	
	# tasks	best scoring competitor			Korn			
Category	supp./all	tool	true	false	true	false	true	false
ControlFlow	19/ 22	CVT-ParPort	15	7	12	7	12	7
Loops	641/ 685	VeriAbs	386	185	80	178	**288**	178
Recursive	57/ 59	UAutomizer	20	18	27	25	27	25
XCSP	109/ 114	CBMC	54	50	46	0	46	† **50**

techniques developed for loops like k-induction have simply not been adapted well to recursive procedures. For Horn clause encodings on the other hand both abstractions are uniform and solvers are largely agnostic to the purpose of predicates. As a downside of enforcing modular proofs, Korn is currently unable to compete in category Arrays, where finding the quantified invariants is hard but state-space exploration succeeds on tasks with fixed loop bounds.

Unfortunately, in the 2023 competition, Eldarica did not run at all due to some unknown problem with the verifier archive, such that Korn terminated way too early and missed out on many results. Table 1 presents results from re-running the evaluation on the competition hardware. This produces 208 additional proofs from Eldarica in category Loops with a hypothetical score of 755 wrt. 323 in SV-COMP 2023, albeit the actual score would be lower than that because usually not all witnesses are confirmed.

4 Software Project, Configuration & Participation

The implementation of Korn is available at https://github.com/gernst/korn under the MIT license, installation instructions are part of the README. The SV-COMP 2023 submission was packaged from commit 8e968dd and shows version 0.4. The included solvers are Z3 4.11.2 64 bit (default configuration) and Eldarica v2.0.8 (using -portfolio). The command line in SV-COMP 2023 is

```
./run -write -model -witness witness.graphml -confirm \
      -random 10 -timeout 20 -z3 -timeout 900 -eld:portfolio <file.c>
```

Participation: ControlFlow, Loops, Recursive, XCSP for ReachSafety.

Contributors. Korn is developed and maintained by the author. G. Alexandru [1] and J. Blau have contributed insights to approach of loop contracts [7].

Data Availability Statement

The tool archive packaged for SV-COMP 2023 is part of the official tools artifact [4] and also available separately [9]. The official competition results [3] are complemented with our post-competition evaluation, based on commit `92e6732` and are available at [8].

References

1. Alexandru, G.: Specifying loops with contracts (2019), Bachelor's Thesis, LMU Munich
2. Beyer, D.: Competition on software verification and witness validation: SV-COMP 2023. In: Proc. TACAS (2). LNCS , Springer (2023)
3. Beyer, D.: Results of the 12th Intl. Competition on Software Verification (SV-COMP 2023). Zenodo (2023). https://doi.org/10.5281/zenodo.7627787
4. Beyer, D.: Verifiers and validators of the 12th Intl. Competition on Software Verification (SV-COMP 2023). Zenodo (2023). https://doi.org/10.5281/zenodo.7627829
5. Bjørner, N., Gurfinkel, A., McMillan, K., Rybalchenko, A.: Horn clause solvers for program verification. In: Fields of Logic and Computation II, pp. 24–51. Springer (2015)
6. Bjørner, N., McMillan, K., Rybalchenko, A.: On solving universally quantified Horn clauses. In: International Static Analysis Symposium. pp. 105–125. Springer (2013)
7. Ernst, G.: Loop verification with invariants and summaries. In: Proc. of Verification, Model-Checking, and Abstract Interpretation (VMCAI). LNCS, vol. 13182. Springer (2022)
8. Ernst, G.: Korn post-competition evaluation. Zenodo (2023). https://doi.org/10.5281/zenodo.7647533
9. Ernst, G.: Korn tool archive as submitted to SV-COMP 2023. Zenodo (2023). https://doi.org/10.5281/zenodo.7647511
10. Ernst, G.: A complete approach to loop verification with invariants and summaries (2020), `https://arxiv.org/abs/2010.05812`, draft
11. Esen, Z., Rümmer, P.: TriCera: Verifying C Programs Using the Theory of Heaps. In: Formal Methods in Computer-aided Design (FMCAD). p. 380 (2022)
12. Gurfinkel, A., Bjørner, N.: The science, art, and magic of Constrained Horn Clauses. In: 2019 21st International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC). pp. 6–10. IEEE (2019)
13. Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The seahorn verification framework. In: Computer Aided Verification. pp. 343–361. Springer (2015)
14. Hehner, E.C.: Specified blocks. In: Working Conference on Verified Software: Theories, Tools, and Experiments. pp. 384–391. Springer (2005)
15. Hojjat, H., Rümmer, P.: The Eldarica Horn solver. In: 2018 Formal Methods in Computer Aided Design (FMCAD). pp. 1–7. IEEE (2018)
16. Tuerk, T.: Local reasoning about while-loops. VSTTE **2010**, 29 (2010)