The Golem Horn Solver

Martin Blicha[1,2][0000−0001−8140−4098], Konstantin Britikov[1][0009−0005−7843−7290], and Natasha Sharygina[1][0000−0002−8872−4913]

[1] Università della Svizzera italiana, Lugano, Switzerland
[2] Charles University, Prague, Czech Republic
{blichm,britik,sharygin}@usi.ch

Abstract. The logical framework of Constrained Horn Clauses (CHC) models verification tasks from a variety of domains, ranging from verification of safety properties in transition systems to modular verification of programs with procedures. In this work we present Golem, a flexible and efficient solver for satisfiability of CHC over linear real and integer arithmetic. Golem provides flexibility with modular architecture and multiple back-end model-checking algorithms, as well as efficiency with tight integration with the underlying SMT solver. This paper describes the architecture of Golem and its back-end engines, which include our recently introduced model-checking algorithm TPA for deep exploration. The description is complemented by extensive evaluation, demonstrating the competitive nature of the solver.

Keywords: Constrained Horn Clauses · Model Checking

1 Introduction

The framework of *Constrained Horn Clauses* (CHC) has been proposed as a unified, purely logic-based, intermediate format for software verification tasks [33]. CHC provides a powerful way to model various verification problems, such as safety, termination, and loop invariant computation, across different domains like transition systems, functional programs, procedural programs, concurrent systems, and more [33,34,41,35]. The key advantage of CHC is the separation of modelling from solving, which aligns with the important software design principle—*separation of concerns*. This makes CHCs highly reusable, allowing a specialized CHC solver to be used for different verification tasks across domains and programming languages. The main focus of the front end is then to translate the source code into the language of constraints, while the back end can focus solely on the well-defined formal problem of deciding satisfiability of a CHC system.

CHC-based *verification* is becoming increasingly popular, with several frameworks developed in recent years, including SeaHorn, Korn and TriCera for C [36,27,28], JayHorn for Java [44], RustHorn for Rust [48], HornDroid for Android [18], SolCMC and SmartACE for Solidity [2,57]. A novel CHC-based approach for *testing* also shows promising results [58]. The growing demand from verifiers drives the development of specialized *Horn* solvers. Different solvers

implement different techniques based on, e.g., model-checking approaches (such as predicate abstraction [32], CEGAR [22] and IC3/PDR [16,26]), machine learning, automata, or CHC transformations. Eldarica [40] uses predicate abstraction and CEGAR as the core solving algorithm. It leverages Craig interpolation [23] not only to guide the predicate abstraction but also for acceleration [39]. Additionally, it controls the form of the interpolants with *interpolation abstraction* [53,46]. Spacer [45] is the default algorithm for solving CHCs in Z3 [51]. It extends PDR-style algorithm for nonlinear CHC [38] with under-approximations and leverages *model-based projection* for predecessor computation. Recently it was enriched with *global guidance* [37]. Ultimate TreeAutomizer [25] implements automata-based approaches to CHC solving [43,56]. HoIce [20] implements a machine-learning-based technique adapted from the ICE framework developed for discovering inductive invariants of transition systems [19]. FreqHorn [29,30] combines syntax-guided synthesis [4] with data derived from unrollings of the CHC system.

According to the results of the international competition on CHC solving CHC-COMP [54,31,24], solvers applying model-checking techniques, namely Spacer and Eldarica, are regularly outperforming the competitors. These are the solvers most often used as the back ends in CHC-based verification projects. However, only specific algorithms have been explored in these tools for CHC solving, limiting their application for diverse verification tasks. Experience from software verification and model checking of transition systems shows that in contrast to the state of affairs in CHC solving, it is possible to build a flexible infrastructure with a unified environment for multiple back-end solving algorithms. CPAchecker [6,7,8,9,10,11], and Pono [47] are examples of such tools.

This work aims to bring this flexibility to the general domain-independent framework of constrained Horn clauses. We present Golem, a new solver for CHC satisfiability, that provides a unique combination of flexibility and efficiency.[3] Golem implements several SMT-based model-checking algorithms: our recent model-checking algorithm based on *Transition Power Abstraction* (TPA) [14,13], and state-of-the-art model-checking algorithms Bounded Model Checking (BMC) [12], k-induction [55], Interpolation-based Model Checking (IMC) [49], Lazy Abstractions with Interpolants (LAWI) [50] and Spacer [45]. Golem achieves efficiency through tight integration with the underlying interpolating SMT solver OpenSMT [17,42] and preprocessing transformations based on *predicate elimination*, *clause merging* and *redundant clause elimination*. The flexible and modular framework of OpenSMT enables customization for different algorithms; its powerful interpolation modules, particularly, offer fine control (in size and strength) with multiple interpolant generation procedures. We report experimentation that confirms the advantage of multiple diverse solving techniques and shows that Golem is competitive with state-of-the-art Horn solvers on large sets of problems.[4] Overall, Golem can serve as an efficient back end for domain-

[3] Golem is available at https://github.com/usi-verification-and-security/golem

specific verification tools and as a research tool for prototyping and evaluating SMT- and interpolation-based verification techniques in a unified setting.

2 Tool overview

In this section, we describe the main components and features of the tool together with the details of its usage. For completeness, we recall the terminology related to CHCs first.

Constrained Horn clauses A constrained Horn clause is formula $\varphi \wedge B_1 \wedge B_2 \wedge \ldots \wedge B_n \implies H$, where φ is the *constraint*, a formula in the background theory, $B_1, \ldots, B_n$ are uninterpreted predicates, and H is an uninterpreted predicate or *false*. The antecedent of the implication is commonly denoted as the *body* and the consequent as the *head*. A clause with more than one predicate in the body is called *nonlinear*. A nonlinear system of CHCs has at least one nonlinear clause; otherwise, the system is linear.

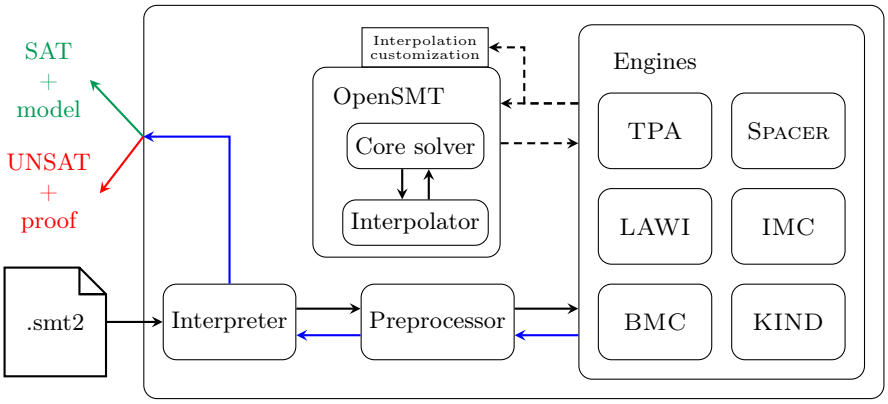


Fig. 1: High-level architecture of Golem

Architecture The flow of data inside Golem is depicted in Fig. 1. The system of CHCs is read from `.smt2` file, a script in an extension of the language of SMT-LIB.[5] `Interpreter` interprets the SMT-LIB script and builds the internal representation of the system of CHCs. In Golem, CHCs are first *normalized*, then the system is translated into an internal graph representation. Normalization rewrites clauses to ensure that each predicate has only variables as arguments.

[4] This is in line with results from CHC-COMP 2021 and 2022 [31,24]. In 2022, Golem beat other solvers except Z3-Spacer in the LRA-TS, LIA-Lin and LIA-Nonlin tracks.

[5] https://chc-comp.github.io/format.html

The graph representation of the system is then passed to the `Preprocessor`, which applies various transformations to simplify the input graph. `Preprocessor` then hands the transformed graph to the chosen back-end engine. Engines in Golem implement various SMT-based model-checking algorithms for solving the CHC satisfiability problem. There are currently six engines in Golem: TPA, BMC, KIND, IMC, LAWI, and Spacer (see details in Sec. 3). User selects the engine to run using a command-line option `--engine`. Golem relies on the interpolating SMT solver OpenSMT [42] not only for answering SMT queries but also for interpolant computation required by most of the engines. Interpolating procedures in OpenSMT can be customized on demand for the specific needs of each engine [1]. Additionally, Golem re-uses the data structures of OpenSMT for representing and manipulating terms.

Models and proofs Besides solving the CHC satisfiability problem, a *witness* for the answer is often required by the domain-specific application. Satisfiability witness is a *model*, an interpretation of the CHC predicates that makes all clauses valid. Unsatisfiability witness is a *proof*, a derivation of the empty clause from the input clauses. In software verification these witnesses correspond to program invariants and counterexample paths, respectively. All engines in Golem produce witnesses for their answer. Witnesses from engines are translated back through the applied preprocessing transformations. Only after this *backtranslation*, the witness matches the original input system and is reported to the user. Witnesses must be explicitly requested with the option `--print-witness`.

Models are internally stored as formulas in the background theory, using only the variables of the (normalized) uninterpreted predicates. They are presented to the user in the format defined by SMT-LIB [5]: a sequence of SMT-LIB's `define-fun` commands, one for each uninterpreted predicate.

For the proofs, Golem follows the trace format proposed by Eldarica. Internally, proofs are stored as a sequence of derivation steps. Every derivation step represents a ground instance of some clause from the system. The ground instances of predicates from the body form the *premises* of the step, and the ground instance of the head's predicate forms the *conclusion* of the step. For the derivation to be valid, the premises of each step must have been derived earlier, i.e., each premise must be a conclusion of some derivation step earlier in the sequence. To the user, the proof is presented as a sequence of derivations of ground instances of the predicates, where each step is annotated with the indices of its premises. See Example 1 below for the illustration of the proof trace.

Golem also implements an internal *validator* that checks the correctness of the witnesses. It validates a model by substituting the interpretations for the predicates and checking the validity of all the clauses with OpenSMT. Proofs are validated by checking all conditions listed above for each derivation step. Validation is enabled with an option `--validate` and serves primarily as a debugging tool for the developers of witness production.

Example 1. Consider the following CHC system and the proof of its unsatisfiability.

$$
\begin{array}{rcl}
x > 0 & \implies & L1(x) \\
x' = x + 1 & \implies & D(x, x') \\
L1(x) \wedge D(x, x') & \implies & L2(x') \\
L2(x) \wedge x \leq 2 & \implies & \mathit{false}
\end{array}
\qquad
\begin{array}{lll}
1.\ L_1(1) & & \\
2.\ D(1,2) & & \\
3.\ L_2(2) & ;1,2 & \\
4.\ \mathit{false} & ;3 &
\end{array}
$$

The derivation of *false* consists of four derivation steps. Step 1 instantiates the first clause for $x := 1$. Step 2 instantiates the second clause for $x := 1$ and $x' := 2$. Step 3 applies resolution to the instance of the third clause for $x := 1$ and $x' := 2$ and facts derived in steps 1 and 2. Finally, step 4 applies resolution to the instance of the fourth clause for $x := 2$ and the fact derived in step 3.

Preprocessing transformations Preprocessing can significantly improve performance by transforming the input CHC system into one more suitable for the back-end engine. The most important transformation in Golem is *predicate elimination*. Given a predicate not present in both the body and the head of the same clause, the predicate can be eliminated by exhaustive application of the resolution rule. This transformation is most beneficial when it also decreases the number of clauses. *Clause merging* is a transformation that merges all clauses with the same uninterpreted predicates in the body and the head to a single clause by disjoining their constraints. This effectively pushes work from the level of the model-checking algorithm to the level of the SMT solver. Additionally, Golem detects and deletes *redundant clauses*, i.e., clauses that cannot participate in the proof of unsatisfiability.

An important feature of Golem is that all applied transformations are *reversible* in the sense that any model or proof for the transformed system can be translated back to a model or proof of the original system.

3 Back-end Engines of Golem

The core components of Golem that solve the problem of satisfiability of a CHC system are referred to as *back-end engines*, or just engines. Golem implements several popular state-of-the-art algorithms from model checking and software verification: BMC, k-induction, IMC, LAWI and Spacer. These algorithms treat the problem of solving a CHC system as a *reachability* problem in the graph representation.

The unique feature of Golem is the implementation of the new model-checking algorithm based on the concept of *Transition Power Abstraction* (TPA). It is capable of much deeper analysis than other algorithms when searching for counterexamples [14], and it discovers *transition* invariants [13], as opposed to the usual (state) invariants.

3.1 Transition Power Abstraction

The TPA engine in Golem implements the model-checking algorithm based on the concept of Transition Power Abstraction. It can work in two modes: The first mode implements the basic TPA algorithm, which uses a single TPA sequence [14]. The second mode implements the more advanced version, split-TPA, which relies on two TPA sequences obtained by splitting the single TPA sequence of the basic version [13]. In Golem, both variants use the under-approximating *model-based projection* for propagating truly reachable states, avoiding full quantifier elimination. Moreover, they benefit from incremental solving available in OpenSMT, which speeds up the satisfiability queries.

The TPA algorithms, as described in the publications, operate on transition systems [14,13]. However, the engine in Golem is not limited to a single transition system. It can analyze a connected *chain of transition systems*. In the software domain, this model represents programs with a sequence of consecutive loops. The extension to the chain of transition systems works by maintaining a separate TPA sequence for each node on the chain, where each node has its own transition relation. The reachable states are propagated forwards on the chain, while safe states—from which final error states are unreachable—are propagated backwards. In this scenario, transition systems on the chain are queried for reachability between various initial and error states. Since the transition relations remain the same, the summarized information stored in the TPA sequences can be re-used across multiple reachability queries. The learnt information summarizing multiple steps of the transition relation is not invalidated when the initial or error states change.

Golem's TPA engine discovers counterexample paths in unsafe transition systems, which readily translate to unsatisfiability proofs for the corresponding CHC systems. For safe transition systems, it discovers safe k-inductive transition invariants. If a model for the corresponding CHC system is required, the engine first computes a quantified inductive invariant and then applies quantifier elimination to produce a quantifier-free inductive invariant, which is output as the corresponding model.[6]

The TPA engine's ability to discover deep counterexamples and transition invariants gives Golem a unique edge for systems requiring deep exploration. We provide an example of this capability as part of the evaluation in Sec. 4.

3.2 Engines for state-of-the-art model-checking algorithms

Besides TPA, Golem implements several popular state-of-the-art model-checking algorithms. Among them are bounded model checking [12], k-induction [55] and McMillan's interpolation-based model checking [49], which operate on transition systems. Golem faithfully follows the description of the algorithms in the respective publications.

[6] The generation of unsatisfiability proofs also works for the extension to chains of transition systems, while the generation of models for this case is still under development.

Additionally, Golem implements *Lazy Abstractions with Interpolants* (LAWI), an algorithm introduced by McMillan for verification of software [50].[7] In the original description, the algorithm operates on programs represented with *abstract reachability graphs*, which map straightforwardly to *linear* CHC systems. This is the input supported by our implementation of the algorithm in Golem.

The last engine in Golem implements the IC3-based algorithm Spacer [45] for solving general, even nonlinear, CHC systems. Nonlinear CHC systems can model programs with summaries, and in this setting, Spacer computes both under-approximating and over-approximating summaries of the procedures to achieve modular analysis of programs. Spacer is currently the only engine in Golem capable of solving nonlinear CHC systems.

All engines in Golem rely on OpenSMT for answering SMT queries, often leveraging the incremental capabilities of OpenSMT to implement the corresponding model-checking algorithm efficiently. Additionally, the engines IMC, LAWI, Spacer and TPA heavily use the flexible and controllable interpolation framework in OpenSMT [52,1], especially multiple interpolation procedures for linear-arithmetic conflicts [3,15].

4 Experiments

In this section, we evaluate the performance of individual Golem's engines on the benchmarks from the latest edition of CHC-COMP. The goal of these experiments is to 1) demonstrate the usefulness of multiple back-end engines and their potential combined use for solving various problems, and 2) compare Golem against state-of-the-art Horn solvers. The benchmark collections of CHC-COMP represent a rich source of problems from various domains.[8] Version 0.3.2 of Golem was used for these experiments. Z3-Spacer (Z3 4.11.2) and Eldarica 2.0.8 were run (with default options) for comparison as the best Horn solvers available. All experiments were conducted on a machine with an AMD EPYC 7452 32-core processor and 8x32 GiB of memory; the timeout was set to 300 seconds. No conflicting answers were observed in any of the experiments. The results are in line with the results of the last editions of CHC-COMP where Golem participated [31,24]. Our artifact for reproducing the experiments is available at https://doi.org/10.5281/zenodo.7973428.

4.1 Category LRA-TS

We ran all engines of Golem on all 498 benchmarks from the LRA-TS (transition systems over linear real arithmetic) category of CHC-COMP.

Table 1 shows the number of benchmarks solved per engine, together with a *virtual best* (VB) engine.[9] On unsatisfiable problems, the differences between the engines' performance are not substantial, but the BMC engine firmly dominates

[7] It is also known as Impact, which was the first tool that implemented the algorithm.
[8] https://github.com/orgs/chc-comp/repositories
[9] Virtual best engine picks the best performance from all engines for each benchmark.

	BMC	KIND	IMC	LAWI	SPACER	SPLIT-TPA	VB
SAT	0	260	145	279	195	128	360
UNSAT	86	84	70	76	69	72	86

Table 1: Number of solved benchmarks from LRA-TS category.

the others. On satisfiable problems, we see significant differences. Fig. 2 plots, for each engine, the number of solved *satisfiable* benchmarks (x-axis) within the given time limit (y-axis, log scale).

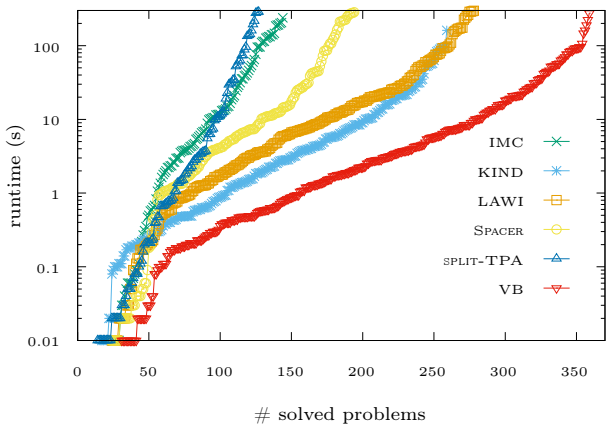


Fig. 2: Performance of GOLEM's engines on SAT problems of LRA-TS category.

The large lead of VB suggests that the solving abilities of the engines are widely complementary. No single engine dominates the others on satisfiable instances. The *portfolio* of techniques available in GOLEM is much stronger than any single one of them.

Moreover, the unified setting enables direct comparison of the algorithms. For example, we can conclude from these experiments that the extra check for k-inductive invariants on top of the BMC-style search for counterexamples, as implemented in the KIND engine, incurs only a small overhead on unsatisfiable problems, but makes the KIND engine very successful in solving satisfiable problems.

4.2 Category LIA-Lin

Next, we considered the LIA-Lin category of CHC-COMP. These are linear systems of CHCs with linear integer arithmetic as the background theory. There are many benchmarks in this category, and for the evaluation at the competition, a subset of benchmarks is selected (see [31,24]). We evaluated the LAWI and

Spacer engines of Golem (the engines capable of solving general linear CHC systems) on the benchmarks selected at CHC-COMP 2022 and compared their performance to Z3-Spacer and Eldarica. Notably, we also examined a specific subcategory of LIA-lin, namely `extra-small-lia`[10] with benchmarks that fall into the fragment accepted by Golem's TPA engine.

There are 55 benchmarks in `extra-small-lia` subcategory, all satisfiable, but known to be highly challenging for all tools. The results, given in Table 2, show that split-TPA outperforms not only LAWI and Spacer engines in Golem, but also Z3-Spacer. Only Eldarica solves more benchmars. We ascribe this to split-TPA's capability to perform deep analysis and discover transition invariants.

Golem				
split-TPA	LAWI	Spacer	Z3-Spacer	Eldarica
22	12	18	18	36

Table 2: Number of solved benchmarks from `extra-small-lia` subcategory.

For the whole LIA-Lin category, 499 benchmarks were selected in the 2022 edition of CHC-COMP [24]. The performance of the LAWI and Spacer engines of Golem, Z3-Spacer and Eldarica on this selection is summarized in Table 3. Here, the Spacer engine of Golem significantly outperforms the LAWI engine. Moreover, even though Golem loses to Z3-Spacer, it beats Eldarica. Given that Golem is a prototype, and Z3-Spacer and Eldarica have been developed and optimized for several years, this demonstrates the great potential of Golem.

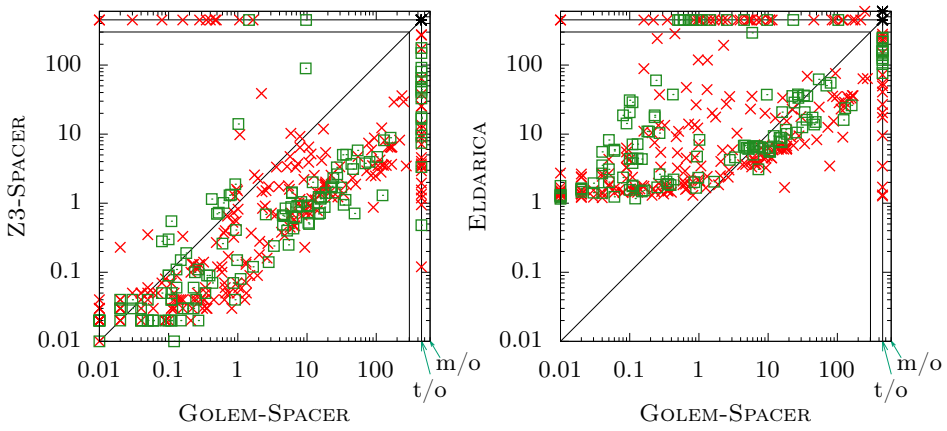
	Golem			
	LAWI	Spacer	Z3-Spacer	Eldarica
SAT	131	184	211	183
UNSAT	77	82	96	60

Table 3: Number of solved benchmarks from LIA-Lin category.

4.3 Category LIA-Nonlin

Finally, we considered the LIA-Nonlin category of benchmarks of CHC-COMP, which consists of *nonlinear* systems of CHCs with linear integer arithmetic as the background theory. For the experiments, we used the 456 benchmarks selected for the 2022 edition of CHC-COMP. Spacer is the only engine in Golem capable of solving nonlinear CHC systems; thus, we focused on a more detailed comparison of its performance against Z3-Spacer and Eldarica. The results of the experiments are summarized in Fig. 3 and Table 4.

[10] https://github.com/chc-comp/extra-small-lia



(a) Golem vs Z3-Spacer (b) Golem vs Eldarica

Fig. 3: Comparison on LIA-Nonlin category (× - SAT, ⊡ - UNSAT).

	Golem-Spacer	Z3-Spacer	Eldarica
SAT	239 (4)	248 (13)	221 (6)
UNSAT	124 (2)	139 (5)	122 (0)

Table 4: Number of solved benchmarks from LIA-Nonlin category. The number of *uniquely* solved benchmarks is in parentheses.

Overall, Golem solved fewer problems than Z3-Spacer but more than Eldarica; however, *all* tools solved some instances *uniquely*. A detailed comparison is depicted in Fig. 3. For each benchmark, its data point in the plot reflects the runtime of Golem (x-axis) and the runtime of the competitor (y-axis). The plots suggest that the performance of Golem is often orthogonal to Eldarica, but highly correlated with the performance of Z3-Spacer. This is not surprising as the Spacer engine in Golem is built on the same core algorithm. Even though Golem is often slower than Z3-Spacer, there is a non-trivial amount of benchmarks on which Z3-Spacer times out, but which Golem solves fairly quickly. Thus, Golem, while being a newcomer, already complements existing state-of-the-art tools, and more improvements are expected in the near future.

To summarise, the overall experimentation with different engines of Golem demonstrates the advantages of the multi-engine general framework and illustrates the competitiveness of its analysis. It provides a lot of flexibility in addressing various verification problems while being easily customizable with respect to the analysis demands.

5 Conclusion

In this work, we presented Golem, a flexible and effective Horn solver with multiple back-end engines, including recently-introduced TPA-based model-checking

algorithms. Golem is a suitable research tool for prototyping new SMT-based model-checking algorithms and comparing algorithms in a unified framework. Additionally, the effective implementation of the algorithm achieved with tight coupling with the underlying SMT solver makes it an efficient back end for domain-specific verification tools. Future directions for Golem include support for VMT input format [21] and analysis of liveness properties, extension of TPA to nonlinear CHC systems, and support for SMT theories of arrays, bit-vectors and algebraic datatypes.

Acknowledgements This work was partially supported by Swiss National Science Foundation grant 200021_185031 and by Czech Science Foundation Grant 23-06506 S.

References

1. Alt, L.: Controlled and Effective Interpolation. Ph.D. thesis, Università della Svizzera italiana (2016), available at https://susi.usi.ch/usi/documents/318933
2. Alt, L., Blicha, M., Hyvärinen, A.E.J., Sharygina, N.: SolCMC: Solidity compiler's model checker. In: Shoham, S., Vizel, Y. (eds.) Computer Aided Verification. pp. 325–338. Springer International Publishing, Cham (2022)
3. Alt, L., Hyvärinen, A.E.J., Sharygina, N.: LRA interpolants from no man's land. In: Strichman, O., Tzoref-Brill, R. (eds.) Hardware and Software: Verification and Testing. pp. 195–210. Springer International Publishing, Cham (2017)
4. Alur, R., Bodik, R., Juniwal, G., Martin, M.M.K., Raghothaman, M., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis. In: 2013 Formal Methods in Computer-Aided Design. pp. 1–8 (2013)
5. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB Standard: Version 2.6. Tech. rep., Department of Computer Science, The University of Iowa (2017), available at www.SMT-LIB.org
6. Beyer, D., Wendler, P.: Algorithms for software model checking: Predicate abstraction vs. Impact. In: 2012 Formal Methods in Computer-Aided Design (FMCAD). pp. 106–113 (Oct 2012)
7. Beyer, D., Dangl, M.: Software verification with PDR: An implementation of the state of the art. In: Biere, A., Parker, D. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 3–21. Springer International Publishing, Cham (2020)
8. Beyer, D., Dangl, M., Wendler, P.: Boosting k-induction with continuously-refined invariants. In: Kroening, D., Păsăreanu, C.S. (eds.) Computer Aided Verification. pp. 622–640. Springer International Publishing, Cham (2015)
9. Beyer, D., Dangl, M., Wendler, P.: A unifying view on SMT-based software verification. Journal of Automated Reasoning **60**(3), 299–335 (Mar 2018)
10. Beyer, D., Keremoglu, M.E.: CPAchecker: A tool for configurable software verification. In: Gopalakrishnan, G., Qadeer, S. (eds.) Computer Aided Verification. pp. 184–190. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
11. Beyer, D., Lee, N.Z., Wendler, P.: Interpolation and SAT-based model checking revisited: Adoption to software verification. Tech. Rep. 2208.05046, arXiv/CoRR (August 2022)

12. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, W.R. (ed.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 193–207. Springer Berlin Heidelberg, Berlin, Heidelberg (1999)
13. Blicha, M., Fedyukovich, G., Hyvärinen, A.E.J., Sharygina, N.: Split transition power abstractions for unbounded safety. In: Griggio, A., Rungta, N. (eds.) Proceedings of the 22nd Conference on Formal Methods in Computer-Aided Design – FMCAD 2022. pp. 349–358. TU Wien Academic Press (2022). https://doi.org/10.34727/2022/isbn.978-3-85448-053-2_42
14. Blicha, M., Fedyukovich, G., Hyvärinen, A.E.J., Sharygina, N.: Transition power abstractions for deep counterexample detection. In: Fisman, D., Rosu, G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 524–542. Springer International Publishing, Cham (2022)
15. Blicha, M., Hyvärinen, A.E.J., Kofroň, J., Sharygina, N.: Using linear algebra in decomposition of Farkas interpolants. International Journal on Software Tools for Technology Transfer **24**(1), 111–125 (2022)
16. Bradley, A.R.: SAT-based model checking without unrolling. In: Jhala, R., Schmidt, D. (eds.) Verification, Model Checking, and Abstract Interpretation. pp. 70–87. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
17. Bruttomesso, R., Pek, E., Sharygina, N., Tsitovich, A.: The OpenSMT solver. In: Esparza, J., Majumdar, R. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 150–153. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
18. Calzavara, S., Grishchenko, I., Maffei, M.: HornDroid: Practical and sound static analysis of android applications by SMT solving. In: 2016 IEEE European Symposium on Security and Privacy. pp. 47–62 (2016)
19. Champion, A., Chiba, T., Kobayashi, N., Sato, R.: ICE-based refinement type discovery for higher-order functional programs. In: Beyer, D., Huisman, M. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 365–384. Springer International Publishing, Cham (2018)
20. Champion, A., Kobayashi, N., Sato, R.: HoIce: An ICE-based non-linear Horn clause solver. In: Ryu, S. (ed.) Programming Languages and Systems. pp. 146–156. Springer International Publishing, Cham (2018)
21. Cimatti, A., Griggio, A., Tonetta, S.: The VMT-LIB language and tools (2021)
22. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) Computer Aided Verification. pp. 154–169. Springer Berlin Heidelberg, Berlin, Heidelberg (2000)
23. Craig, W.: Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. The Journal of Symbolic Logic **22**(3), 269–285 (1957)
24. De Angelis, E., Vediramana Krishnan, H.G.: CHC-COMP 2022: Competition report. Electronic Proceedings in Theoretical Computer Science **373**, 44–62 (nov 2022)
25. Dietsch, D., Heizmann, M., Hoenicke, J., Nutz, A., Podelski, A.: Ultimate TreeAutomizer (CHC-COMP tool description). In: Angelis, E.D., Fedyukovich, G., Tzevelekos, N., Ulbrich, M. (eds.) Proceedings of the Sixth Workshop on Horn Clauses for Verification and Synthesis and Third Workshop on Program Equivalence and Relational Reasoning, HCVS/PERR@ETAPS 2019, Prague, Czech Republic, 6-7th April 2019. EPTCS, vol. 296, pp. 42–47 (2019)
26. Een, N., Mishchenko, A., Brayton, R.: Efficient implementation of property directed reachability. In: Proceedings of the International Conference on Formal Methods in Computer-Aided Design. pp. 125–134. FMCAD '11, FMCAD Inc, Austin, TX (2011)

27. Ernst, G.: Korn—software verification with Horn clauses (competition contribution). In: Sankaranarayanan, S., Sharygina, N. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 559–564. Springer Nature Switzerland, Cham (2023)
28. Esen, Z., Rümmer, P.: TriCera: Verifying C programs using the theory of heaps. In: Griggio, A., Rungta, N. (eds.) Proceedings of the 22nd Conference on Formal Methods in Computer-Aided Design – FMCAD 2022. pp. 360–391. TU Wien Academic Press (2022)
29. Fedyukovich, G., Kaufman, S.J., Bodík, R.: Sampling invariants from frequency distributions. In: 2017 Formal Methods in Computer Aided Design (FMCAD). pp. 100–107 (2017)
30. Fedyukovich, G., Prabhu, S., Madhukar, K., Gupta, A.: Solving constrained Horn clauses using syntax and data. In: 2018 Formal Methods in Computer Aided Design (FMCAD). pp. 1–9 (2018)
31. Fedyukovich, G., Rümmer, P.: Competition report: CHC-COMP-21. In: Hojjat, H., Kafle, B. (eds.) Proceedings 8th Workshop on Horn Clauses for Verification and Synthesis, HCVS@ETAPS 2021, Virtual, 28th March 2021. EPTCS, vol. 344, pp. 91–108 (2021)
32. Graf, S., Saidi, H.: Construction of abstract state graphs with PVS. In: Grumberg, O. (ed.) Computer Aided Verification. pp. 72–83. Springer Berlin Heidelberg, Berlin, Heidelberg (1997)
33. Grebenshchikov, S., Lopes, N.P., Popeea, C., Rybalchenko, A.: Synthesizing software verifiers from proof rules. In: Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 405–416. PLDI '12, Association for Computing Machinery, New York, NY, USA (2012)
34. 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 (2019)
35. Gurfinkel, A.: Program verification with constrained Horn clauses (invited paper). In: Shoham, S., Vizel, Y. (eds.) Computer Aided Verification. pp. 19–29. Springer International Publishing, Cham (2022)
36. Gurfinkel, A., Kahsai, T., Komuravelli, A., Navas, J.A.: The SeaHorn verification framework. In: Kroening, D., Păsăreanu, C.S. (eds.) Computer Aided Verification. pp. 343–361. Springer International Publishing, Cham (2015)
37. Hari Govind, V.K., Chen, Y., Shoham, S., Gurfinkel, A.: Global guidance for local generalization in model checking. In: Lahiri, S.K., Wang, C. (eds.) Computer Aided Verification. pp. 101–125. Springer International Publishing, Cham (2020)
38. Hoder, K., Bjørner, N.: Generalized property directed reachability. In: Cimatti, A., Sebastiani, R. (eds.) Theory and Applications of Satisfiability Testing – SAT 2012. pp. 157–171. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
39. Hojjat, H., Iosif, R., Konečný, F., Kuncak, V., Rümmer, P.: Accelerating interpolants. In: Chakraborty, S., Mukund, M. (eds.) Automated Technology for Verification and Analysis. pp. 187–202. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
40. Hojjat, H., Rümmer, P.: The Eldarica Horn solver. In: FMCAD. pp. 158–164. IEEE (10 2018)
41. Hojjat, H., Rümmer, P., Subotic, P., Yi, W.: Horn clauses for communicating timed systems. Electronic Proceedings in Theoretical Computer Science **169**, 39–52 (dec 2014)
42. Hyvärinen, A.E.J., Marescotti, M., Alt, L., Sharygina, N.: OpenSMT2: An SMT solver for multi-core and cloud computing. In: Creignou, N., Le Berre, D. (eds.)

Theory and Applications of Satisfiability Testing – SAT 2016. pp. 547–553. Springer International Publishing, Cham (2016)
43. Kafle, B., Gallagher, J.P.: Tree automata-based refinement with application to Horn clause verification. In: D'Souza, D., Lal, A., Larsen, K.G. (eds.) Verification, Model Checking, and Abstract Interpretation. pp. 209–226. Springer Berlin Heidelberg, Berlin, Heidelberg (2015)
44. Kahsai, T., Rümmer, P., Sanchez, H., Schäf, M.: Jayhorn: A framework for verifying Java programs. In: Chaudhuri, S., Farzan, A. (eds.) Computer Aided Verification. pp. 352–358. Springer International Publishing, Cham (2016)
45. Komuravelli, A., Gurfinkel, A., Chaki, S.: SMT-based model checking for recursive programs. Formal Methods in System Design **48**(3), 175–205 (Jun 2016)
46. Leroux, J., Rümmer, P., Subotić, P.: Guiding Craig interpolation with domain-specific abstractions. Acta Informatica **53**(4), 387–424 (2016)
47. Mann, M., Irfan, A., Lonsing, F., Yang, Y., Zhang, H., Brown, K., Gupta, A., Barrett, C.: Pono: A flexible and extensible SMT-based model checker. In: Silva, A., Leino, K.R.M. (eds.) Computer Aided Verification. pp. 461–474. Springer International Publishing, Cham (2021)
48. Matsushita, Y., Tsukada, T., Kobayashi, N.: RustHorn: CHC-based verification for Rust programs. ACM Trans. Program. Lang. Syst. **43**(4) (oct 2021)
49. McMillan, K.L.: Interpolation and SAT-based model checking. In: Computer Aided Verification. pp. 1–13. Springer, Berlin Heidelberg (2003)
50. McMillan, K.L.: Lazy abstraction with interpolants. In: Ball, T., Jones, R.B. (eds.) Computer Aided Verification. pp. 123–136. Springer Berlin Heidelberg, Berlin, Heidelberg (2006)
51. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
52. Rollini, S.F., Alt, L., Fedyukovich, G., Hyvärinen, A.E.J., Sharygina, N.: PeRIPLO: A framework for producing effective interpolants in SAT-based software verification. In: McMillan, K., Middeldorp, A., Voronkov, A. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning. pp. 683–693. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
53. Rümmer, P., Subotić, P.: Exploring interpolants. In: 2013 Formal Methods in Computer-Aided Design. pp. 69–76 (Oct 2013)
54. Rümmer, P.: Competition report: CHC-COMP-20. Electronic Proceedings in Theoretical Computer Science **320**, 197–219 (Aug 2020)
55. Sheeran, M., Singh, S., Stålmarck, G.: Checking safety properties using induction and a SAT-solver. In: Hunt, W.A., Johnson, S.D. (eds.) Formal Methods in Computer-Aided Design. pp. 127–144. Springer Berlin Heidelberg, Berlin, Heidelberg (2000)
56. Wang, W., Jiao, L.: Trace Abstraction Refinement for Solving Horn Clauses. The Computer Journal **59**(8), 1236–1251 (08 2016)
57. Wesley, S., Christakis, M., Navas, J.A., Trefler, R., Wüstholz, V., Gurfinkel, A.: Verifying solidity smart contracts via communication abstraction in smartace. In: Finkbeiner, B., Wies, T. (eds.) Verification, Model Checking, and Abstract Interpretation. pp. 425–449. Springer International Publishing, Cham (2022)
58. Zlatkin, I., Fedyukovich, G.: Maximizing branch coverage with constrained Horn clauses. In: Fisman, D., Rosu, G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 254–272. Springer International Publishing, Cham (2022)