Mode-based Transformation from Satisfiability of Constrained Horn Clauses to Test-friendly Reachability Problem (Extended Abstract)

1 Introduction

The satisfiability of constrained Horn clauses (CHCs) is an important problem, to which various verification problems can be reduced. In this paper, we are interested in improving the efficiency of disproving (the satisfiability of) CHCs. Quickly disproving CHCs is especially important in recent approaches to temporal property verification [7, 8], where a verification problem is underapproximated by CHCs and the approximation is repeatedly refined when the CHCs are found to be unsatisfiable.

Despite recent developments of CHC solvers, the efficiency of disproving CHCs is still not satisfactory. For example, Property-Directed Reachability (PDR) [3, 5, 6] and their variants like Spacer [10] often struggle to find a deep counterexample that requires a large number of iterations over transition relations. To remedy this, Blicha et al. [2] proposed a method to exponentially reduce the number of approximations for disproving an instance compared with PDR and their variants. We think, however, there is still room for improvement in disproving CHCs as we confirm later in our preliminary experiments.

We propose a reduction from CHC satisfiability checking problems to safety property problems of "test-friendly" programs. By doing so, we can apply testing methods directly to the translated program, and make most use of random testing methods, which have been extensively developed recently. One may think that since CHCs are often used to prove safety property problems of programs, we can just apply testing to the original programs. However, since some CHCs are generated from different problems like temporal verification, accessing the original programs is not always possible.

In this paper, we first introduce a "naive" transformation for readers to understand the basic idea and a main challenge of our approach for generating test-friendly programs. Then, we briefly introduce a transformation guided by *mode analyses* to remedy the problem.

2 Preliminary

We use P as the metavariable for predicate variables, v for variables, and $\mathbf{a}$ for arithmetic expressions. We define *body formula* φ as

$$\varphi \quad ::= \quad \theta \mid P(\tilde{\mathbf{a}_i}) \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \exists x.\, \varphi \mid \textbf{if } \theta \textbf{ then } \varphi_1 \textbf{ else } \varphi_2.$$

Though **if** θ **then** φ_1 **else** φ_2 can be expanded to $(\neg\theta \vee \varphi_1) \wedge (\theta \vee \varphi_2)$, we explicitly add this notation for simplicity. A definite clause is a triple of a predicate variable P, a sequence of variables $v_1, \ldots, v_k$ and a body formula, written as $\varphi \Rightarrow P(\tilde{v_i})$. We assume that for each definite clause $\varphi \Rightarrow P(v_1, \ldots, v_k)$, $\mathrm{fv}(\varphi) \subset \{v_1, \ldots, v_k\}$. We call a tuple of a *goal formula* $\varphi \Rightarrow \bot$ and a sequence of definite clauses D whose predicate variables are disjoint a *CHC system*. We also use a subset of OCaml as a target language.

Submitted to:
HCVS 2024

3 A Naive Transformation

Our goal is to transform CHC systems to nondeterministic programs so that the systems are satisfiable if and only if the transformed programs will not raise an uncaught exception. We first define a naive transformation $[\![\varphi]\!]$, a map from a body formula to an expression in the target language [1], inductively as

$$\begin{aligned}
[\![\theta]\!] &= \texttt{assert}(\neg\theta) \\
[\![\varphi_1 \wedge \varphi_2]\!] &= \texttt{try } [\![\varphi_1]\!] \texttt{ with } [\![\varphi_2]\!] \\
[\![\varphi_1 \vee \varphi_2]\!] &= [\![\varphi_1]\!] \star [\![\varphi_2]\!] \\
[\![\exists x.\, \varphi]\!] &= \texttt{let x = rand_int () in } [\![\varphi]\!] \\
[\![\textbf{if } \theta \textbf{ then } \varphi_1 \textbf{ else } \varphi_2]\!] &= \texttt{if } \theta \texttt{ then } [\![\varphi_1]\!] \texttt{ else } [\![\varphi_2]\!] \\
[\![p(\mathbf{a}_1,\ldots,\mathbf{a}_n)]\!] &= p(\mathbf{a}_1,\ldots,\mathbf{a}_n).
\end{aligned}$$

Basically, $[\![\varphi]\!]$ is an expression that raises an exception when φ is true. For a constraint formula, $[\![\theta]\!]$ raises an exception when θ holds. For a conjunction $\varphi_1 \wedge \varphi_2$, when both of $[\![\varphi_1]\!]$ and $[\![\varphi_2]\!]$ raise an uncaught exception, $[\![\varphi_1 \wedge \varphi_2]\!]$ also does. On the other hand, for a disjunction $\varphi_1 \vee \varphi_2$, we execute $[\![\varphi_1]\!]$ and $[\![\varphi_2]\!]$ nondeterministically since it is enough to check if one of them raises an exception. Note that since $[\![\varphi_1]\!]$ or $[\![\varphi_2]\!]$ may result in an infinite loop, to preserve the reachability to an invalid state, we do not execute them sequentially but nondeterministically. We also randomly choose an integer when we handle existential quantifiers. Using this map, we transform a definite clause $\varphi \Rightarrow P(v_1,\ldots,v_k)$ to a recursive function `let rec p` $v_1 \cdots v_k$ `=` $[\![\varphi]\!]$. For a goal formula $\varphi \Rightarrow \bot$, we have `let main =` $[\![\varphi]\!]$.

For the proof of this concept, we have implemented a preliminary transformer from linear CHCs to programs based on the naive transformation, and evaluated the effectiveness of random testing against the transformed programs by using unsatisfiable instances taken from the unsafe AE-VAL benchmarks in CHC-COMP[2]. The timeout was set to 180 seconds in all the experiments. Our solver successfully proved all 54 instances to be unsatisfiable while Golem [1] was able to solve only 30 instances, and Spacer [10] could solve only 14 instances. One reason for this result is that the instances in the benchmark are intended to have a deep counterexample which requires solvers to execute millions of iterations over transition relations to reach an invalid state. Spacer [10] is not good at finding such long counterexamples, and Golem [1] mitigated the situation but still it is not satisfactory.

However, if we apply this naive transformation to non-linear CHCs, we found that this procedure easily diverged and failed to find a counterexample that Spacer or Golem can. This motivated us to extend, formalize and implement our method further.

4 Motivating Example for Mode-Guided Translation

The main difficulty of the approach based on the naive transformation is the nondeterminism in CHCs, especially in *non-linear* CHCs. Let us illustrate this difficulty by considering the following CHC system:

$$\textbf{if } n \geq 100 \textbf{ then } r = n - 10 \textbf{ else } (\mathrm{mc91}(n+11, m) \wedge \mathrm{mc91}(m, r)) \Rightarrow \mathrm{mc91}(n, r) \tag{1}$$

$$n \leq 100 \wedge r \neq 91 \wedge \mathrm{mc91}(n, r) \Rightarrow \bot \tag{2}$$

If we apply the naive transformation to the CHC system, we obtain the program in Figure 1.(a). Even

[1] `try e1 with _ -> e2` is abbreviated to `try e1 with e2` for simplicity.

[2] https://github.com/chc-comp/aeval-unsafe

```
let rec mc91 n r =
  if n >= 100 then assert(r <> n - 10)
  else
    let m = rand_int () in
    try mc91 (n + 11) m with mc91 m r
let main =
  let n = rand_int () in
  let r = rand_int () in
  try assert(n > 100 || r = 91)
  with mc91 n r
```

(a) A reduced program from CHCs (1) – (2)

```
let rec mc91' n =
  if n >= 100 then n - 10
  else
    let m = mc91' (n + 11) in
    mc91' m
let main =
  let n = rand_int () in
  let r = mc91' n in
  assert(n > 100 || r = 91)
```

(b) A program obtained by our translation

Figure 1: Comparison of the programs obtained by different transformations

though the reachability to an assertion failure is equivalent to the unsatisfiability of the given CHC problem, it is unlikely to find a counterexample using this transformed program just by random testing because every time `mc91` is called with the argument $n < 100$, we have to precisely determine `m`.

The main goal of our mode-guided transformation is to eliminate such nondeterminism as much as possible. For the above CHC system, the program in Figure 1.(b), which we wish to obtain by our mode-guided transformation, still has the same reachability property as the original program, but it is much easier to find a counterexample by random testing since we no longer have to choose `m` in the body of the function `mc91'`. The point of this transformation is that for `mc91` clause, we generate a function that takes an integer and returns an integer, instead of a predicate that take two integers. Then, the temporary variable `m` in the original program is eliminated in the body of `mc91'` since `m` can be uniquely determined by the returned value of `mc91'(n + 11)`. Intuitively, given an arbitrary integer `n`, `mc91'` returns an integer `r` such that $mc91(n,r)$ is true.

To achieve this transformation, we have to analyze which argument of a predicate can be an "output", and which cannot. For mc91 clause, we can regard n as an input and r as an output of the clause. We can observe this by analyzing the clause (1). First, we can uniquely obtain such r just by $n-10$ for the if-branch of (1). For the else-branch, since we assume the second argument of mc91 is an output, m can be determined by $\mathrm{mc91}(n+11,m)$, and from $\mathrm{mc91}(m,r)$, we can obtain r as well.

We have formalized these analyses by introducing *modes*, which have been utilized in the logic programming literature for analyzing input and output arguments of a predicate, for integers and predicates, and proposed a mode-guided transformation. We have also proposed a mode inference algorithm to automate the procedure above. Note that the whole procedure is fully automated; users of our solver do not have to write any annotations to the given CHCs nor the transformed programs.

We are currently working on the implementation and evaluation of our proposed method. As mentioned earlier, the transformed program is intended to be easily connected with testing frameworks such as symbolic execution and random testing, but we have not evaluated how effective it is in practice. Extending our transformation is another direction we are interested in. Our mode translation can be naturally extended to higher-order logics like νHFL [9] or Higher-Order Constrained Horn Clauses [4]. We also plan to introduce polymorphic modes for clauses to make our translation more expressive and effective.

References

[1] Martin Blicha, Konstantin Britikov & Natasha Sharygina (2023): *The Golem Horn Solver*. In Constantin Enea & Akash Lal, editors: *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part II*, *Lecture Notes in Computer Science* 13965, Springer, pp. 209–223, doi:10.1007/978-3-031-37703-7_10. Available at `https://doi.org/10.1007/978-3-031-37703-7_10`.

[2] Martin Blicha, Grigory Fedyukovich, Antti E. J. Hyvärinen & Natasha Sharygina (2022): *Transition Power Abstractions for Deep Counterexample Detection*. In Dana Fisman & Grigore Rosu, editors: *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*, *Lecture Notes in Computer Science* 13243, Springer, pp. 524–542, doi:10.1007/978-3-030-99524-9_29. Available at `https://doi.org/10.1007/978-3-030-99524-9_29`.

[3] Aaron R. Bradley (2011): *SAT-Based Model Checking without Unrolling*. In Ranjit Jhala & David A. Schmidt, editors: *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings*, *Lecture Notes in Computer Science* 6538, Springer, pp. 70–87, doi:10.1007/978-3-642-18275-4_7. Available at `https://doi.org/10.1007/978-3-642-18275-4_7`.

[4] Toby Cathcart Burn, C.-H. Luke Ong & Steven J. Ramsay (2018): *Higher-order constrained horn clauses for verification*. *Proc. ACM Program. Lang.* 2(POPL), pp. 11:1–11:28, doi:10.1145/3158099. Available at `https://doi.org/10.1145/3158099`.

[5] Niklas Eén, Alan Mishchenko & Robert K. Brayton (2011): *Efficient implementation of property directed reachability*. In Per Bjesse & Anna Slobodová, editors: *International Conference on Formal Methods in Computer-Aided Design, FMCAD '11, Austin, TX, USA, October 30 - November 02, 2011*, FMCAD Inc., pp. 125–134. Available at `http://dl.acm.org/citation.cfm?id=2157675`.

[6] Krystof Hoder & Nikolaj Bjørner (2012): *Generalized Property Directed Reachability*. In: *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*, pp. 157–171, doi:10.1007/978-3-642-31612-8_13. Available at `https://doi.org/10.1007/978-3-642-31612-8_13`.

[7] Naoki Kobayashi, Takeshi Nishikawa, Atsushi Igarashi & Hiroshi Unno (2019): *Temporal Verification of Programs via First-Order Fixpoint Logic*. In: *Proceedings of SAS 2019*, pp. 413–436, doi:10.1007/978-3-030-32304-2_20.

[8] Naoki Kobayashi, Kento Tanahashi, Ryosuke Sato & Takeshi Tsukada (2023): *HFL(Z) Validity Checking for Automated Program Verification*. *Proc. ACM Program. Lang.* 7(POPL), pp. 154–184, doi:10.1145/3571199. Available at `https://doi.org/10.1145/3571199`.

[9] Naoki Kobayashi, Takeshi Tsukada & Keiichi Watanabe (2018): *Higher-Order Program Verification via HFL Model Checking*. In: *Proceedings of ESOP 2018*, *LNCS* 10801, Springer, pp. 711–738, doi:10.1007/978-3-319-89884-1_25.

[10] Anvesh Komuravelli, Arie Gurfinkel & Sagar Chaki (2016): *SMT-based model checking for recursive programs*. *Formal Methods in System Design* 48(3), pp. 175–205, doi:10.1007/s10703-016-0249-4.