Interrogation Testing of CHC Solvers

DAVID KAINDLSTORFER, TU Wien, Austria
ANASTASIA ISYCHEV, TU Wien, Austria
VALENTIN WÜSTHOLZ, Diligence Security, Austria
MARIA CHRISTAKIS, TU Wien, Austria

A Constrained Horn Clause (CHC) is a specific type of logic formula that contains uninterpreted predicates. CHC formulas are often used by static program analyzers to encode program properties, which are then verified using CHC solvers. The solvers themselves are complex tools and may contain bugs, which can lead to verifying unsafe programs, flagging safe programs as unsafe, or providing analyzers with incorrect invariants and counterexamples. It is, therefore, crucial to develop techniques for systematically testing CHC solvers.

In this paper, we present the first interrogation-testing technique for CHC solvers, which we implement in a tool called HornGator. Our technique uses witnesses generated by the solver under test to form new CHC instances. It also integrates a knowledge base maintaining a history of past solver queries. All this information helps HornGator generate more diverse instances, thereby improving its bug-finding effectiveness. As a result, HornGator found 21 unique bugs in five state-of-the-art CHC solvers, all of which are confirmed by the developers, 18 are fixed, and eight are of the highest severity.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: interrogation testing, CHC solvers, fuzzing

ACM Reference Format:
David Kaindlstorfer, Anastasia Isychev, Valentin Wüstholz, and Maria Christakis. 2026. Interrogation Testing of CHC Solvers. *Proc. ACM Softw. Eng.* 3, FSE, Article FSE076 (July 2026), 20 pages. https://doi.org/10.1145/3797104

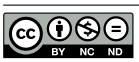
1 Introduction

A *Constrained Horn Clause* (CHC) is a specific type of logic formula that includes *uninterpreted* (i.e., unknown) predicates. A CHC *instance* is a set of CHCs; given an instance as input, a CHC solver tries to instantiate all uninterpreted predicates such that the set of CHCs is satisfiable. Examples of popular CHC solvers are Spacer [22] and Eldarica [16]. Such solvers are widely used in static program analysis and verification [3], for instance, in SeaHorn [15] for LLVM bitcode, JayHorn [17] for Java, and RustHorn [26] for Rust.

Bugs in CHC solvers. CHC solvers are complex tools, which makes it quite likely that they contain bugs in their implementations. Tab. 1 shows the severity of the different types of bugs that may occur in CHC solvers. For bugs of severity 1 (the highest severity), the solver returns Sat (i.e., satisfiable) for an instance that is Unsat (i.e., unsatisfiable). Such bugs may have detrimental consequences; they may render the program analyzers that rely on the CHC solver unsound. As a result, unsafe code could be proved safe by the affected analyzers.

Bugs of severity 2 cause the solver to return Unsat for a Sat instance. This type of bugs may result in incompleteness in program analyzers. In other words, the affected analyzers could flag safe

Authors' Contact Information: David Kaindlstorfer, TU Wien, Vienna, Austria, david.kaindlstorfer@tuwien.ac.at; Anastasia Isychev, TU Wien, Vienna, Austria, anastasia.isychev@tuwien.ac.at; Valentin Wüstholz, Diligence Security, Vienna, Austria, valentin.wustholz@diligence.security; Maria Christakis, TU Wien, Vienna, Austria, maria.christakis@tuwien.ac.at.



ACM 2994-970X/2026/7-ARTFSE076
https://doi.org/10.1145/3797104

Tab. 1. Severity of the different types of bugs that may occur in CHC solvers. GT stands for ground truth and SR for solver response.

GT \ SR	SAT	UNSAT	UNKNOWN	Other
SAT	3a	2	4a	4b
UNSAT	1	3b	4a	4b

code as unsafe. This can, in turn, lead to developers wasting valuable time looking for a non-existent bug in their code.

Bugs can also occur when the solver returns the correct response; see severity 3 in the table. Consider that when an instance is found to be SAT, the solver can typically provide a *model*, that is, an instantiation of all uninterpreted predicates in the instance such that it is satisfiable. However, the model could be incorrect due to a solver bug (severity 3a); in particular, the instantiation of certain predicates might not lead to a satisfiable instance. As a consequence, program analyzers that use CHC solvers for invariant inference (for instance, by using uninterpreted predicates as "placeholders" when capturing the constraints that an inductive loop invariant would have to satisfy) could infer incorrect invariants.

On the other hand, when an instance is found to be UNSAT, the solver typically provides a *refutation proof*, which can be used to construct a counterexample trace. Just like models in the SAT case, refutation proofs may also be incorrect (severity 3b). This could lead to analyzers generating wrong counterexamples for unsafe code, potentially misleading developers when debugging.

Finally, severity 4a refers to cases where the solver returns UNKNOWN for instances that lie in a decidable theory fragment, and severity 4b to solver crashes, insufficient input validation, and other less critical bugs.

Related work. Given the consequences of the more severe bug types, it is crucial to develop techniques for systematically testing CHC solvers.

Why not use differential testing [27]? Differential testing, although widely used, is ill-suited as the primary approach for CHC solvers. These tools differ widely in the theories they support—e.g., only some handle arrays or algebraic data types (ADTs)—as well as in their dialects and accepted syntax. For instance, bug 1 in Sect. 5 involves ADTs unsupported by CHC solvers GOLEM [4], THETA [33], and ULTIMATE TREE AUTOMIZER [13], and bug 2 relies on a solver-specific construct that only ELDARICA understands. In addition, certain high-complexity instances can only be solved by a single tool while others fail to respond; for example, bug 5 can only be handled by ELDARICA. These discrepancies make consistent comparisons across solvers infeasible. More generally, even when solvers can process the same input, divergent answers do not reveal which solver is at fault, undermining the diagnostic value of differential testing.

How about SMT-solver fuzzers (e.g.,[20, 24, 30, 34])? While SMT and CHC solvers share a common input language (SMT-LIB [2]), they solve fundamentally different problems: SMT solvers decide the satisfiability of formulas, whereas CHC solvers synthesize interpretations for uninterpreted predicates that satisfy a system of Horn clauses. For instance, invariant synthesis—a core verification task—can be reduced to CHC solving, but not to SMT solving. This distinction introduces fundamental challenges for reusing SMT fuzzers, as CHC solvers differ not only in the types of inputs they consume but also in the outputs they produce.

In terms of inputs, although both classes of solvers support SMT-LIB, they do so in mutually incompatible ways. SMT fuzzers typically generate formulas without uninterpreted predicates, making them inadequate for exercising CHC-specific logic. As an analogy, SMT fuzzers are to CHC fuzzers what C-compiler fuzzers are to Java-compiler fuzzers: some Java code may also be valid C,

but effectively testing Java requires handling different syntax and semantics. Similarly, SMT-level mutations often violate the structural constraints imposed by CHC solvers, leading to syntactically invalid inputs.

In terms of outputs, CHC solvers may provide predicate models or refutation proofs in addition to Sat/Unsat. SMT solvers also return models, but these typically assign concrete values to first-order variables; likewise, when available, SMT proof objects differ substantially in structure from CHC refutation proofs. The CHC-specific outputs are central to HornGator's design as they drive its transformations and are essential for bug finding. Moreover, SMT fuzzers tend to focus on refutational soundness bugs—returning Unsat for satisfiable inputs—which are considered the most severe from a program-analysis perspective. In contrast, the most critical bugs in CHC solvers involve incorrectly returning Sat for unsatisfiable systems, a class of errors that certain SMT fuzzers cannot detect (e.g., [24]).

Given these limitations, there is a clear need for dedicated techniques to test CHC solvers that do not rely on differential testing. Recent work proposed HornFuzz [31], a technique based on *metamorphic testing* [12] that is able to detect bugs of severity 1, 2, and 3a. HornFuzz was used to test Spacer; it detected two bugs of severity 2 and 13 bugs of severity 3a, where the generated model was incorrect. However, as we will see, HornFuzz relies primarily on SMT-style rewrites (e.g., reordering conjuncts) and fails to uncover several severe bugs. To the best of our knowledge, there has been no other dedicated attempt to test CHC solvers.

Our approach. In this paper, we present the first interrogation-testing technique for CHC solvers. *Interrogation testing* [18] is a metamorphic-testing approach that can be applied to a wide range of analysis tools. Its core principles—leveraging analysis witnesses and maintaining a knowledge base—are domain independent. Witnesses, such as inferred invariants or proofs, provide information beyond the basic analysis response and can be systematically used to generate new, semantically related analysis queries. The knowledge base serves as a persistent repository of queries, responses, and associated witnesses, enabling the tester to generate increasingly diverse and targeted test cases. In prior work, interrogation testing was applied to program analyzers that reason about reachability.

We instantiate interrogation testing for CHC solvers for the first time. This requires designing CHC-specific transformations, as CHC solvers produce fundamentally different artifacts—namely, models for predicates and refutation proofs—compared to the reachability witnesses used in prior work. Our instantiation leverages both types of solver feedback—models and proofs—going beyond simple Sat or Unsat responses, and uses the knowledge base to generate new CHC instances with known satisfiability. These instances are both semantically meaningful and syntactically diverse, making them more effective for exposing bugs in CHC solvers.

We implement our technique in a tool called HornGator and use it to test five CHC solvers that participate in CHC-Comp [1], namely Eldarica [16], Golem [4], Spacer [22], Theta [33], and Ultimate Tree Automizer [13]. Our tool focuses on detecting bugs of severity 1–3; bugs of lower severity are less critical and easy to find. In particular, bugs of severity 4a can be found by passing instances of decidable theory fragments (e.g., from CHC-Comp) to the solver and reporting a bug whenever the solver returns Unknown. Bugs of severity 4b can be found by generating random instances, even non-CHC ones, and passing them to the solver.

HornGator found a total of 21 unique bugs in all solvers we tested. All bugs were confirmed by the developers, and 18 of them are already fixed. Eight bugs have severity 1, ten have severity 2, and the remaining three have severity 3. As shown by our experimental evaluation, our bugs would not have been found or would have been found with a significant slowdown without the two core principles of interrogation testing. We also show that HornGator is significantly more effective in bug finding than HornFuzz.

Our contributions. Our paper makes the following contributions:

- We present an interrogation-testing technique for detecting severe bugs in CHC solvers.
- We implement our technique in the open-source tool HornGator.
- We evaluate the effectiveness of HornGator by testing five solvers that participate in CHC-Comp. Our results significantly push the state of the art.

Outline. Next, we provide brief background on CHCs. In Sect. 3, we give an overview of HornGator, and in Sect. 4, we explain the technical details of our approach. We present our experimental evaluation in Sect. 5, discuss related work in Sect. 6, and conclude in Sect. 7.

2 Background: Constrained Horn Clauses

A CHC *instance* is a set of CHCs. Each CHC is a first-order logic formula of the following form [14]:

$$\forall V \cdot \varphi \wedge p_1(X_1) \wedge \ldots \wedge p_n(X_n) \rightarrow h(X)$$

Here, V denotes a set of variables and $X_1, \ldots, X_n, X$ are terms over V. φ denotes a constraint over a background theory, such as linear arithmetic, arrays, bit-vectors, etc., $p_1, \ldots, p_n$ are uninterpreted predicates, and h is an uninterpreted predicate or $\bot$ (i.e., *false*). Without loss of generality, let us assume that φ is a conjunction of sub-constraints $\varphi_1 \wedge \ldots \wedge \varphi_m$. Note that this conjunction may express any φ.

An instance is *satisfiable* if there exists at least one instantiation of all uninterpreted predicates in the instance such that the set of CHCs holds. As an example, consider the following CHC instance:

$$\forall x \cdot x \leq 0 \rightarrow P(x) \tag{1}$$

$$\forall x, y \cdot P(x) \wedge x < 2 \wedge y = x + 1 \rightarrow P(y) \tag{2}$$

$$\forall x \cdot P(x) \wedge x \geq 3 \rightarrow \bot \tag{3}$$

This set of clauses is satisfiable, and a model for the uninterpreted predicate is: $P(x) = \{x \mid x < 3\}$.

An instance is *unsatisfiable* if there is no instantiation of the uninterpreted predicates in the instance such that the set of CHCs holds. For example, when changing clause (3) to

$$\forall x \cdot P(x) \wedge x \geq 2 \rightarrow \bot \tag{4}$$

the set of clauses becomes unsatisfiable. The refutation proof derives $\bot$ as follows:

$$\top \xrightarrow{(1)} P(0) \xrightarrow{(2)} P(1) \xrightarrow{(2)} P(2) \xrightarrow{(3)} \bot$$

It says that, if $x = 0$, we know that $P(0)$ holds from clause (1). Given $P(0)$, the left-hand side of clause (2) holds ($P(0) \wedge 0 < 2 \wedge y = 0 + 1$), and thus, clause (2) establishes $P(1)$ on the right-hand side. Given $P(1)$, clause (2) can now also establish $P(2)$, i.e., $P(1) \wedge 1 < 2 \wedge y = 1 + 1 \rightarrow P(2)$. In turn, $P(2)$ makes the left-hand side of clause (4) hold, i.e., $P(2) \wedge 2 \geq 2$, which derives $\bot$. In this proof, $P(0)$, $P(1)$, and $P(2)$ are called *proof facts* [4].

For simplicity, we omit the outermost universal quantifier in the rest of this paper, and we consider all free variables to be universally quantified.

3 Overview

Fig. 1 shows an overview of HornGator. Given a CHC *seed instance*—our implementation supports multiple seed instances—and a CHC solver under test, the *interrogator* component initially queries the solver with the seed instance. The seed instance could be taken from CHC-Comp, the solver's test suite, or it may be provided by the user. In our experiments, we use instances from CHC-Comp.

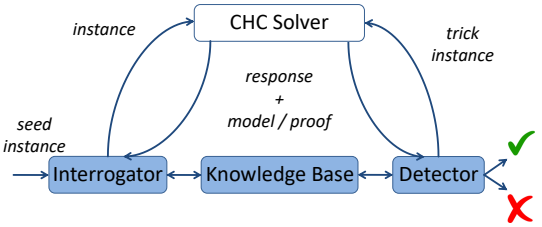


Fig. 1. Overview of HornGator.

When querying the solver with an instance, the interrogator requests not only the solver response (e.g., Sat or Unsat), but also the model in case of Sat and the refutation proof in case of Unsat. Let us refer to the model or refutation proof as the *witness*.

If the solver does not timeout or return Unknown, then the CHC instance is suitable for detecting solver bugs of severity 1–3. In this case, the instance together with the solver response and the corresponding witness are stored in the *knowledge base* (partially inspired by how greybox fuzzers use a corpus of interesting inputs). Such an interrogation round may be repeated any number of times to gradually enrich the knowledge base. The interrogator may also use the knowledge base to construct new CHC instances (by mutating existing ones) with which to query the solver.

At any point, the *detector* component may query the solver with a *trick instance*. A trick instance is a CHC instance constructed from information in the knowledge base such that the expected solver response is known a-priori. In other words, the detector uses one or more instances from the knowledge base together with the corresponding solver response and witness to generate a new instance whose satisfiability is known in advance. In Sect. 4, we introduce different categories of transformations that HornGator applies to derive trick queries. The transformations, and especially how they use the response and witness, are one of the key novelties of our approach.

If the solver response for the trick instance contradicts the expected response, HornGator reports a bug. The bug severity is determined through manual inspection as we discuss later. If no bug is detected and the trick instance is suitable for further testing (i.e., no Unknown response or timeout), the instance and all relevant information are added to the knowledge base.

Example bug. Starting from an existing instance from CHC-Comp'23 (in the "linear integer arithmetic with nonlinear clauses" category), HornGator detected a bug of severity 1 in the Golem solver (bug 9 in Tab. 2). In particular, the solver returned Sat for the following unsatisfiable instance:

$$\begin{aligned} x < y &\rightarrow P(x, y) \\ P(x, 1) &\rightarrow Q \\ Q \wedge P(x, 7) \wedge x > y &\rightarrow R(y) \\ R(3) &\rightarrow \bot \end{aligned}$$

Note that the original instance from CHC-Comp is unsatisfiable and contains almost 18K LOC. The detector eventually reports the above instance (minimized both automatically and manually for presentation purposes) after repeatedly querying the solver with instances generated using the knowledge base. To find this buggy instance, HornGator applied transformations that replace conjuncts on the left-hand side of a clause with $\top$ (i.e., *true*); as we explain later, when applied to an Unsat instance, such a transformation preserves its satisfiability (i.e., the transformed instance should also be Unsat). In addition, HornGator dropped clauses that do not appear in the refutation proof, and therefore, are not needed by the solver to derive Unsat.

This bug was quickly fixed by the Golem developers, who thanked us and encouraged us to send them more such issues.

4 Transformations

In this section, we describe in detail the transformations that the interrogator and detector may apply to instances stored in the knowledge base to generate new ones. Note that the interrogator can apply any transformation to any instance (regardless of its satisfiability); its goal is to enrich the knowledge base with diverse instances, which is crucial for exploring different solver components and optimization paths. The detector, on the other hand, carefully applies transformations such that the solver response for the transformed instances can be anticipated and a bug can be detected in case of a contradiction.

The detector categorizes the transformations as follows:

- **Sat-to-Sat.** These transformations are applied to a satisfiable instance and ensure that the transformed instance is also satisfiable.
- **Unsat-to-Unsat.** These transformations are applied to an unsatisfiable instance and ensure that the transformed instance is also unsatisfiable.
- **Satisfiability preserving.** These transformations are applied to any instance, satisfiable or unsatisfiable, and ensure that the transformed instance preserves the satisfiability of the original instance. In particular, if the original instance is satisfiable (resp. unsatisfiable), the transformed instance is also satisfiable (resp. unsatisfiable).
- **Instance fusion.** These transformations fuse two instances into one, and the satisfiability of the newly generated instance depends on the satisfiability of the original instances.

4.1 Sat-to-Sat Transformations

We have designed the following five transformations for this category.

PlugModelLHS. Given a Sat instance, this transformation first chooses a clause $\varphi \wedge p_1 \wedge \ldots \wedge p_n \rightarrow h$ and a left-hand-side (LHS) uninterpreted predicate p_i, where $i \in \{1, ..., n\}$. It then replaces p_i with its instantiation from the model, denoted by $M(p_i)$, obtaining

$$\varphi \wedge \boldsymbol{M(p_i)} \wedge p_1 \wedge \ldots \wedge p_{i-1} \wedge p_{i+1} \wedge \ldots \wedge p_n \rightarrow h.$$

Since the original instance is Sat, there must exist a model M that provides instantiations for all uninterpreted predicates such that the CHCs hold. Intuitively, M is still a model for any transformed instance that instantiates *some* of the uninterpreted predicates. Consequently, when replacing an uninterpreted predicate with one of these instantiations—specifically, from the model generated by the CHC solver—the transformed instance should remain satisfiable.

As an example, consider the clause

$$x \leq 5 \wedge \boldsymbol{P(x, y)} \wedge Q(y) \rightarrow R(x),$$

where $M(P) = x \geq y$. When applying this transformation, we obtain the clause

$$x \leq 5 \wedge \boldsymbol{x \geq y} \wedge Q(y) \rightarrow R(x).$$

Note that in this and all following transformations, we highlight how the formulas are transformed in bold.

PlugModelRHS. Given a Sat instance, this transformation chooses a clause $\varphi \wedge p_1 \wedge \ldots \wedge p_n \rightarrow h$ and replaces the right-hand-side (RHS) uninterpreted predicate h with its instantiation from the model, denoted by $M(h)$. To preserve the CHC syntax, we rewrite the clause to

$$\varphi \wedge \boldsymbol{\neg M(h)} \wedge p_1 \wedge \ldots \wedge p_n \rightarrow \bot.$$

The above clause is equivalent to the implication

$$\varphi \wedge p_1 \wedge \ldots \wedge p_n \rightarrow \boldsymbol{M(h)},$$

which however does not adhere to the CHC syntax. They are equivalent because of the implication property

$$(A \rightarrow B) \equiv (A \wedge \neg B \rightarrow \mathit{false}).$$

As an example, consider the clause

$$x \leq 5 \wedge P(x, y) \wedge Q(y) \rightarrow \boldsymbol{R(x)},$$

where $M(R) = x \geq 0$. When applying this transformation, we obtain the clause

$$x \leq 5 \wedge \boldsymbol{\neg(x \geq 0)} \wedge P(x, y) \wedge Q(y) \rightarrow \boldsymbol{\bot}.$$

AddCstrLHS. Given a Sat instance, this transformation chooses a clause $\varphi \wedge p_1 \wedge \ldots \wedge p_n \rightarrow h$ and conjoins any constraint φ_{m+1} to $\varphi \equiv \varphi_1 \wedge \ldots \wedge \varphi_m$, potentially including $\bot$. Note that φ_{m+1} needs to be over the same background theory as φ.

The transformation strengthens the LHS of a satisfiable implication $(A \rightarrow B) \equiv (\neg A \vee B)$. Since the original disjunction $\neg A \vee B$ holds, the transformed disjunction $\neg(A \wedge \varphi_{m+1}) \vee B = \neg\varphi_{m+1} \vee (\neg A \vee B)$ should also hold.

As an example consider again the clause

$$x \leq 5 \wedge P(x, y) \wedge Q(y) \rightarrow R(x).$$

A possible transformed clause could be obtained by adding the conjunct $x \neq y$:

$$x \leq 5 \wedge \boldsymbol{x \neq y} \wedge P(x, y) \wedge Q(y) \rightarrow R(x)$$

UnplugLHS. Given a Sat instance, this transformation chooses a conjunct φ_j, where $j \in \{1, ..., m\}$, occurring in $\varphi \equiv \varphi_1 \wedge \ldots \wedge \varphi_m$ of a clause $\varphi \wedge p_1 \wedge \ldots \wedge p_n \rightarrow h$. It then replaces φ_j with a fresh uninterpreted predicate p_{fresh} (of appropriate arity). More specifically, given a clause

$$\varphi_1 \wedge \ldots \wedge \varphi_{j-1} \wedge \boldsymbol{\varphi_j} \wedge \varphi_{j+1} \wedge \ldots \wedge \varphi_m \wedge p_1 \wedge \ldots \wedge p_n \rightarrow h,$$

we obtain

$$\varphi_1 \wedge \ldots \wedge \varphi_{j-1} \wedge \varphi_{j+1} \wedge \ldots \wedge \varphi_m \wedge p_1 \wedge \ldots \wedge p_n \wedge \boldsymbol{p_{fresh}} \rightarrow h.$$

The intuition here is that the transformed instance should remain satisfiable because there exists a model for p_{fresh}, namely φ_j or, trivially, $\bot$.

As an example, consider the clause

$$\boldsymbol{x \geq 5} \rightarrow Q(x),$$

where $\varphi_j = x \geq 5$. When applying this transformation, we obtain

$$\boldsymbol{P(x)} \rightarrow Q(x).$$

UnplugRHS. Given a Sat instance, this transformation chooses a conjunct φ_j, where $j \in \{1, ..., m\}$, occurring in $\varphi \equiv \varphi_1 \wedge \ldots \wedge \varphi_m$ of a clause $\varphi \wedge p_1 \wedge \ldots \wedge p_n \rightarrow \bot$. Note that the right-hand side is $\bot$. It then removes φ_j from the left-hand side and adds a fresh uninterpreted predicate p_{fresh} (of appropriate arity) to the right-hand side of the clause. More specifically, given a clause

$$\varphi_1 \wedge \ldots \wedge \varphi_{j-1} \wedge \boldsymbol{\varphi_j} \wedge \varphi_{j+1} \wedge \ldots \wedge \varphi_m \wedge p_1 \wedge \ldots \wedge p_n \rightarrow \boldsymbol{\bot},$$

we obtain

$$\varphi_1 \wedge \ldots \wedge \varphi_{j-1} \wedge \varphi_{j+1} \wedge \ldots \wedge \varphi_m \wedge p_1 \wedge \ldots \wedge p_n \wedge \rightarrow \boldsymbol{p_{fresh}}.$$

The intuition here is that the transformed instance should remain satisfiable because there exists a model for p_{fresh}, namely $\neg\varphi_j$ or, trivially, $\top$. The negation in $\neg\varphi_j$ is explained by the implication property

$$(A \rightarrow B) \equiv (A \wedge \neg B \rightarrow \mathit{false}).$$

As an example, consider the clause

$$x \neq 5 \wedge \boldsymbol{\neg(x \geq y)} \wedge Q(y) \rightarrow \boldsymbol{\bot},$$

where $\varphi_j = \neg(x \geq y)$. When applying this transformation, we obtain the clause

$$x \neq 5 \wedge Q(y) \rightarrow \boldsymbol{P(x, y)},$$

where a model for P is $x \geq y$.

Summary. Transformations PlugModelLHS and PlugModelRHS transform an instance using both the solver response, i.e., that the original instance is found to be Sat, and the model. The remaining transformations rely only on the solver response.

Transformation AddCstrLHS essentially strengthens the left-hand side of a satisfiable clause, which as we explained, preserves its satisfiability. The other transformations either instantiate an uninterpreted predicate (PlugModelLHS, PlugModelRHS) or uninstantiate a constraint (UnplugLHS, UnplugRHS).

4.2 Unsat-to-Unsat Transformations

We have designed the following four transformations for this category.

PlugTopLHS. Given an Unsat instance, this transformation chooses a clause $\varphi \wedge p_1 \wedge \ldots \wedge p_n \rightarrow h$ and replaces a conjunct φ_j, where $j \in \{1, ..., m\}$, in $\varphi \equiv \varphi_1 \wedge \ldots \wedge \varphi_m$ or a LHS uninterpreted predicate p_i, where $i \in \{1, ..., n\}$, with $\top$. More specifically, when replacing a conjunct φ_j, we obtain

$$\varphi_1 \wedge \ldots \wedge \varphi_{j-1} \wedge \boldsymbol{\top} \wedge \varphi_{j+1} \wedge \ldots \wedge \varphi_m \wedge p_1 \wedge \ldots \wedge p_n \rightarrow h.$$

Similarly, when replacing a predicate p_i, we obtain

$$\varphi \wedge p_1 \wedge \ldots \wedge p_{i-1} \wedge \boldsymbol{\top} \wedge p_{i+1} \wedge \ldots \wedge p_n \rightarrow h.$$

This transformation weakens the LHS of an unsatisfiable implication $(A \rightarrow B) \equiv (\neg A \vee B)$. In our case, A is a conjunction, and thus, $\neg A$ is a disjunction of the negated conjuncts. By applying our transformation, a disjunct in $\neg A$ becomes *false*. So, if the original disjunction does not hold, the transformed disjunction should also not hold.

For example, consider the unsatisfiable instance from Sect. 2:

$$\begin{aligned} x \leq 0 &\rightarrow P(x) \\ P(x) \wedge \boldsymbol{x < 2} \wedge y = x + 1 &\rightarrow P(y) \\ P(x) \wedge x \geq 2 &\rightarrow \bot \end{aligned}$$

When replacing conjunct $x < 2$ with $\top$, we obtain:

$$\begin{aligned} x \leq 0 &\rightarrow P(x) \\ P(x) \wedge \boldsymbol{\top} \wedge y = x + 1 &\rightarrow P(y) \\ P(x) \wedge x \geq 2 &\rightarrow \bot \end{aligned}$$

This transformation was also used to detect the Golem bug described in Sect. 3.

PlugBottomRHS. Given an Unsat instance, this transformation chooses a clause $\varphi \wedge p_1 \wedge \ldots \wedge p_n \rightarrow h$ and replaces h with $\bot$. We, therefore, obtain

$$\varphi \wedge p_1 \wedge \ldots \wedge p_n \rightarrow \boldsymbol{\bot}.$$

The intuition here is similar to that for the PlugTopLHS transformation. In particular, we strengthen the RHS of an implication $(A \rightarrow B) \equiv (\neg A \vee B)$ by replacing B with *false*. So, if the original disjunction does not hold, the transformed disjunction should also not hold.

As an example, consider again the unsatisfiable instance from Sect. 2 (also shown above). When replacing the RHS of the second clause, i.e., $P(y)$, with $\bot$, we obtain:

$$\begin{aligned} x \leq 0 &\rightarrow P(x) \\ P(x) \wedge x < 2 \wedge y = x + 1 &\rightarrow \bot \\ P(x) \wedge x \geq 2 &\rightarrow \bot \end{aligned}$$

DropRedundantClause. Given an Unsat instance, this transformation chooses a clause that does not appear in the refutation proof and removes it.

If a clause does not appear in the refutation proof, then it is not necessary to prove that the set of CHCs is unsatisfiable. Consequently, removing it from the set should not affect the solver response—it should still be Unsat. Note that, instead of simply removing it, we could also apply any transformation to a clause not appearing in the refutation proof and still ensure that the transformed instance is Unsat.

As an example, consider the clauses:

$$x < y \rightarrow P(x, y) \tag{5}$$
$$P(x, 1) \rightarrow Q \tag{6}$$
$$Q \wedge P(x, 7) \wedge x > y \rightarrow R(y) \tag{7}$$
$$\boldsymbol{y < 0 \wedge P(x, y) \rightarrow R(y)} \tag{8}$$
$$R(3) \rightarrow \bot \tag{9}$$

A refutation proof for this instance is:

$$\left.\begin{array}{l} \top \xrightarrow{(5)} P(0,1) \xrightarrow{(6)} Q \\ \top \xrightarrow{(5)} P(4,7) \end{array}\right\} \xrightarrow{(7)} R(3) \xrightarrow{(9)} \bot$$

It says that, if $x = 0, y = 1$, we know that $P(0, 1)$ holds from clause (5). The same clause also establishes $P(4, 7)$ if $x = 4, y = 7$. Given $P(0, 1)$, the left-hand side of clause (6) holds, and thus, it establishes Q on the right-hand side. Given Q and $P(4, 7)$, if $y = 3$, the left-hand side of clause (7) holds, and as a result, $R(3)$ holds. Finally, given $R(3)$, clause (9) derives $\bot$. Now, since clause (8) is not used in the refutation proof, applying the above transformation results in retaining only the other clauses:

$$\begin{aligned} x < y &\rightarrow P(x, y) \\ P(x, 1) &\rightarrow Q \\ Q \wedge P(x, 7) \wedge x > y &\rightarrow R(y) \\ R(3) &\rightarrow \bot \end{aligned}$$

As we briefly mentioned in Sect. 3, this transformation was used to detect the Golem bug of severity 1. Specifically, for the above transformed instance, Golem returned Sat instead of Unsat.

ReplaceClauseWithFact. Given an Unsat instance, this transformation selects a fact $F = q(\dots)$ from the refutation proof, where q is an uninterpreted predicate. The transformation then replaces the clause that the refutation proof uses to derive F by $\top \rightarrow F$.

The intuition here is that any clauses that lead to the derivation of F are no longer needed. So, this transformation effectively eliminates the first part of the proof establishing F, states that F

holds with clause $\top \rightarrow F$, and given F, it expects the solver to show that the transformed instance is also unsatisfiable.

As an example, consider again the unsatisfiable instance from Sect. 2 and its refutation proof. Let us select fact $F = P(0)$ derived using clause (1). This transformation, therefore, replaces clause (1) by $\top \rightarrow P(0)$ and generates the following (also unsatisfiable) instance:

$$\begin{aligned} \boldsymbol{\top \rightarrow P(0)} \\ P(x) \wedge x < 2 \wedge y = x + 1 \rightarrow P(y) \\ P(x) \wedge x \geq 2 \rightarrow \bot \end{aligned}$$

Summary. Transformations DropRedundantClause and ReplaceClauseWithFact transform an instance using both the solver response, i.e., that the original instance is found to be Unsat, and the refutation proof. The remaining transformations rely only on the solver response.

PlugTopLHS weakens the left-hand side of an unsatisfiable clause, which as we explained, preserves its satisfiability. Similarly, PlugBottomRHS strengthens the right-hand side of an unsatisfiable clause. DropRedundantClause removes a clause that is not used in the refutation proof, whereas ReplaceClauseWithFact replaces a clause that is used to derive a fact from the proof with the fact.

4.3 Satisfiability-Preserving Transformations

The following transformation preserves the satisfiability of any instance, that is, satisfiable, resp. unsatisfiable, instances should remain satisfiable, resp. unsatisfiable.

UnplugLHS+Clause. This transformation is similar to UnplugLHS except that it may also be applied to unsatisfiable instances. In comparison to UnplugLHS, it additionally introduces a clause $\varphi_j \rightarrow p_{fresh}$.

If the original instance is satisfiable, its model may be extended for the transformed instance with an interpretation of p_{fresh}, namely φ_j. Conversely, if the original instance is unsatisfiable, its refutation proof may extended for the transformed instance with p_{fresh} facts.

For example, consider the unsatisfiable instance from Sect. 2 and its refutation proof. When replacing conjunct $y = x + 1$ with $p_{fresh}(x, y)$, we obtain:

$$x \leq 0 \rightarrow P(x) \tag{10}$$

$$P(x) \wedge x < 2 \wedge \boldsymbol{p_{fresh}(x, y)} \rightarrow P(y) \tag{11}$$

$$P(x) \wedge x \geq 2 \rightarrow \bot \tag{12}$$

$$\boldsymbol{y = x + 1 \rightarrow p_{fresh}(x, y)} \tag{13}$$

The refutation proof now becomes:

$$\left.\begin{array}{l} \top \xrightarrow{(10)} P(0) \\ \top \xrightarrow{(13)} p_{fresh}(0,1) \\ \top \xrightarrow{(13)} p_{fresh}(1,2) \end{array}\right\} \xrightarrow{(11)} P(1) \left.\vphantom{\begin{array}{l} \top \xrightarrow{(10)} P(0) \\ \top \xrightarrow{(13)} p_{fresh}(0,1) \\ \top \xrightarrow{(13)} p_{fresh}(1,2) \end{array}}\right\} \xrightarrow{(11)} P(2) \xrightarrow{(12)} \bot$$

4.4 Instance-Fusion Transformations

These transformations fuse two instances into one, and the satisfiability of the newly generated instance depends on the satisfiability of the original instances.

FuseStrong. This transformation fuses two CHC instances as follows. Assume C_1 and C_2 are the clauses in each instance. The fused instance contains clauses $C_1 \cup C_2$. We guarantee (by

renaming) that the predicate names in C_1 and C_2 are disjoint. By conjoining the two sets of clauses, we ensure that the fused instance is satisfiable if both original instances are satisfiable, otherwise it is unsatisfiable.

As an example, consider clauses C_1 to be

$$\begin{aligned} x \leq 5 \wedge P(x,y) \wedge Q(y) &\rightarrow R(x) \\ x = 0 \wedge R(0) &\rightarrow \bot, \end{aligned}$$

which are satisfiable, and clauses C_2 (from Sect. 3) to be

$$\begin{aligned} x < y &\rightarrow P(x,y) \\ P(x,1) &\rightarrow Q \\ Q \wedge P(x,7) \wedge x > y &\rightarrow R(y) \\ R(3) &\rightarrow \bot, \end{aligned}$$

which are unsatisfiable. The fused instance is then

$$\begin{aligned} x \leq 5 \wedge P(x,y) \wedge Q(y) &\rightarrow R(x) \\ x = 0 \wedge R(0) &\rightarrow \bot \\ x < y &\rightarrow P'(x,y) \\ P'(x,1) &\rightarrow Q' \\ Q' \wedge P'(x,7) \wedge x > y &\rightarrow R'(y) \\ R'(3) &\rightarrow \bot, \end{aligned}$$

and it is unsatisfiable.

FuseWeak. This transformation fuses two CHC instances as follows. Assume C_1 and C_2 are the clauses in each instance. The transformation replaces all clauses of the form $\varphi \wedge p_1 \wedge \ldots \wedge p_n \rightarrow \bot$ with $\varphi \wedge p_1 \wedge \ldots \wedge p_n \rightarrow f_1$ in C_1 and with $\varphi \wedge p_1 \wedge \ldots \wedge p_n \rightarrow f_2$ in C_2, where f_1 and f_2 are fresh uninterpreted predicates. It then adds the clause $f_1 \wedge f_2 \rightarrow \bot$ expressing that if both instances are unsatisfiable, the fused instance is also unsatisfiable, otherwise it is satisfiable. Again, we guarantee (by renaming) that the predicate names in C_1 and C_2 are disjoint.

As an example, consider clauses C_1 and C_2 from the above transformation. The fused instance is then

$$\begin{aligned} x \leq 5 \wedge P(x,y) \wedge Q(y) &\rightarrow R(x) \\ x = 0 \wedge R(0) &\rightarrow \boldsymbol{F_1} \\ x < y &\rightarrow P'(x,y) \\ P'(x,1) &\rightarrow Q' \\ Q' \wedge P'(x,7) \wedge x > y &\rightarrow R(y) \\ R'(3) &\rightarrow \boldsymbol{F_2} \\ \boldsymbol{F_1 \wedge F_2} &\boldsymbol{\rightarrow \bot}, \end{aligned}$$

and it is satisfiable.

4.5 Transformation Combinations

Of course, transformations may be stacked, that is, we can apply multiple transformations to a given instance at a time as long as they agree on the expected solver response for the transformed instance.

Within this constraint, the selection and application order of transformations is randomized to maximize diversity.

More interestingly, however, HornGator can combine certain transformations in a more meaningful way. In particular, regarding the UnplugLHS and UnplugLHS+Clause transformations, if φ_j occurs in constraint φ of more than one clause in the instance, we can replace it multiple times using the *same* uninterpreted predicate p_{fresh}. Moreover, if $\neg\varphi_j$ occurs in clauses where $h \equiv \bot$, we can also replace it with the same p_{fresh} appearing on the right-hand side (see UnplugRHS).

5 Experimental Evaluation

We evaluate HornGator by testing five CHC solvers that participate in CHC-Comp, namely Eldarica, Golem, Spacer, Theta, and Ultimate Tree Automizer. In our evaluation, we address the following research questions:

RQ1: How effective is HornGator in detecting bugs of different severity in CHC solvers?
RQ2: What are characteristics of the detected bugs?
RQ3: How efficient is HornGator?
RQ4: Which transformations were effective in bug detection?
RQ5: How effective is interrogation testing vs. metamorphic testing?
RQ6: How does HornGator compare to HornFuzz?

5.1 Experimental Setup

Testing time. We implemented an early version of HornGator and started testing Eldarica and Spacer. We then improved our tool, designed more transformations, and added support for more solvers. Each solver was tested between one and six months. During the testing time, we did not report any duplicate bugs. Whenever a bug was detected, we either paused testing the affected solver until a fix was provided by the developers, or we manually de-duplicated any additional bugs before reporting them.

Fuzzing campaigns. We define two setups for our fuzzing campaigns, namely the *bug-detection* and *time-to-bug* setups.

The bug-detection setup was used to find previously unknown bugs in the solvers under test during the above testing time. In this setup, the interrogator initializes the knowledge base with five random seed instances from CHC-Comp. We use all 2,336 instances from the 2023 edition of the competition as well as all 300 instances from the newly introduced category of "ADTs with arrays" in the 2024 edition. The knowledge base is reset after 100 queries to the solver, and following each reset, the interrogator uses new random seed instances to re-initialize the knowledge base. We use a solver timeout of 60s per instance.

The time-to-bug setup is used to measure the time it takes HornGator to re-find fixed bugs. For this setup, the interrogator initializes the knowledge base only with those seed instances from CHC-Comp that were previously needed to expose the bug. Note that these instances are at most two—all our transformations are applied to a single instance, except for the fusion ones, which are applied to two. The knowledge base is again reset after 100 queries to the solver, and following each reset, the interrogator uses the same seed instances to re-initialize the knowledge base. We again use a solver timeout of 60s per instance and introduce a timeout of 72h for the fuzzing campaign. To ensure reproducibility of the results, each fuzzing campaign is controlled by a numeric random seed. To account for the randomness in HornGator, we run our time-to-bug experiments with 20 different numeric random seeds.

Hardware. We performed all experiments on a machine with two AMD EPYC 9474F CPUs @ 3.60GHz and 1.5TB of memory, running Debian GNU/Linux 12 (bookworm).

Tab. 2. Bugs detected by HornGator.

Bug ID	CHC Solver	Bug Severity				Bug Status
		1	2	3a	3b	
1	Eldarica		✓			Fixed
2	Eldarica			✓		Fixed
3	Eldarica		✓			Fixed
4	Eldarica			✓		Fixed
5	Eldarica		✓			Fixed
6	Eldarica		✓			Fixed
7	Golem		✓			Fixed
8	Golem	✓				Fixed
9	Golem	✓				Fixed
10	Golem		✓	✓	✓	Fixed
11	Golem		✓			Fixed
12	Spacer		✓	✓		Fixed
13	Spacer	✓	✓			Fixed
14	Spacer	✓	✓			Fixed
15	Spacer	✓	✓			Fixed
16	Spacer	✓				Fixed
17	Theta		✓			Fixed
18	U. T. Automizer		✓			Fixed
A	Eldarica	✓				Confirmed
B	Spacer			✓		Confirmed
C	Spacer	✓				Confirmed

5.2 Experimental Results

We now discuss our findings for each research question.

RQ1: Effectiveness of HornGator. Tab. 2 shows the list of unique bugs detected by HornGator using the bug-detection setup. The first column provides an identifier for each bug and links to the (anonymized) bug report on GitHub. The second column shows the CHC solver for which the bug was found. The next two columns indicate the bug severity (as explained in Tab. 1) and its status (confirmed or fixed). We use numbers to identify fixed bugs and letters for confirmed ones.

As shown in the table, a bug may have different severity classifications. This is because there are multiple ways in which it might affect the solver, and therefore manifest itself. Bug severity is determined through manual inspection, and of course, by confirming with the solver developers. In the table, we show all the ways in which each bug manifested itself during our experiments. For example, due to bug 10, Golem erroneously returned Unsat for the satisfiable instance of Fig. 2b, but there were also cases where this bug caused the solver to return an invalid model (severity 3a) or refutation proof (severity 3b).

The lower bug count in Theta and U. T. Automizer is primarily due to two factors. First, these solvers are significantly slower than others, which reduces test throughput and limits the number of explored inputs during the fuzzing campaign. Second, they support fewer theories compared to the other solvers, which restricts the number of suitable seed input files. Additionally, during our testing period, the issues in these solvers took a long time to be acknowledged and fixed, so we were unable to continue fuzzing them effectively—we repeatedly encountered the same bug without further progress.

$$x < y \to P(x, y)$$
$$x < y \wedge P(y, z) \to Q(x, z)$$
$$Q(1, 3) \to R_1$$
$$Q(0, 2) \to R_2$$
$$R_1 \wedge R_2 \to \bot$$

(a) Bug 8 in Golem.

$$(x \wedge y) \wedge \neg x \wedge P(1) \to P(4)$$
$$P(2) \to \bot$$

(b) Bug 10 in Golem.

$$\top \to P(0, x, y, y)$$
$$(\forall(e\ E).(e = c)) \wedge \neg(x = 0 \wedge u = z) \wedge P(x, y, z, u) \to \bot$$

(c) Bug 1 in Eldarica.

Fig. 2. Bugs found by HornGator in Golem and Eldarica.

In total, HornGator detected 21 unique bugs, all of which are confirmed by the developers and 18 are fixed. Considering only the highest severity of each bug, HornGator detected eight bugs of severity 1, ten bugs of severity 2, and three bugs of severity 3a. We found bugs in all solvers we tested and of all severity classifications we considered.

RQ2: Detected bugs. Here, we discuss sample bugs found by HornGator in Golem, Eldarica, and Spacer, which were fixed by the developers.

Fig. 2a shows an instance that revealed bug 8 of severity 1 in Golem. The instance is unsatisfiable because the first two clauses establish $Q(1, 3)$ and $Q(0, 2)$. Nevertheless, Golem returned Sat. The bug was due to incorrect handling of variable names—upon parsing, auxiliary variables are prefixed with "aux", but this was not done when such variables were added during preprocessing. Interestingly, the buggy code contained the comment "*Fixme: This is hackish*". This bug could only be revealed with our DropRedundantClause and ReplaceClauseWithFact transformations.

Fig. 2b shows an instance that revealed bug 10 of severity 2 in Golem. The instance is trivially satisfiable since the left-hand side of the first clause evaluates to false. This bug was found in the solver's preprocessing component, which incorrectly classified the instance as trivially unsatisfiable.

Fig. 2c shows an instance that revealed bug 1 of severity 2 in Eldarica. It uses algebraic data types (ADTs) to define an enum E (not shown in the figure), which only has value c. Given this, $\forall(e\ E).(e = c)$ on the left-hand side of the last clause is a tautology. Spacer returned Unsat for this satisfiable instance due to a bug related to quantifier elimination in the context of ADTs. This bug could only be revealed with our PlugModelLHS transformation. In particular, formula $\forall(e\ E).(e = c)$ is part of a model returned by the solver. Note that plugging in the entire model, i.e., also instantiating P, does not reveal the bug.

Bug 12 of severity 2 in Spacer was found when the solver returned Unsat for a satisfiable instance. The bug was due to Spacer's slicing procedure, which aggressively sliced away relevant parts of the instance.

Overall, we received very positive developer feedback for our bug reports. We already mentioned in Sect. 3 that the Golem developers encouraged us to report more bugs. The Eldarica developers called bugs 2 and 3 a "*great catch*", and for bug 4, they said: "*The bug must have been in the solver since around 2010, so this is an awesome catch.*"

RQ3: Efficiency of HornGator. To evaluate our tool's efficiency, we measure the time it takes HornGator to re-find the bugs it previously detected. For this experiment, we use the time-to-bug setup and include only fixed bugs to ensure that a detected bug is indeed the one we intended to re-find. In particular, for each detected bug, we automatically apply the fix proposed by the solver developers and check that the bug disappears.

We exclude bugs 3 and 4 from this experiment as HornGator cannot re-find them within the time limit of 72h. These bugs are very intricate to detect and were found after weeks of testing

Tab. 3. Average time for HornGator and its baselines to detect a fixed bug.

Bug ID	HornGator			
	w/o kb&wtn	w/o kb	w/o wtn	full
1	–	02h 56m 40s	–	**00h 28m 27s**
2	16h 10m 57s	17h 30m 04s	06h 50m 30s	**06h 39m 12s**
5	*33h 31m 59s*	13h 21m 29s	12h 40m 00s	**02h 12m 16s**
6	01h 09m 08s	01h 37m 19s	00h 07m 59s	**00h 07m 02s**
8	–	**00h 00m 51s**	–	00h 01m 05s
9	–	00h 02m 48s	–	**00h 01m 49s**
10	00h 04m 42s	**00h 01m 00s**	00h 14m 49s	00h 07m 47s
11	–	–	**07h 17m 44s**	12h 09m 56s
12	02h 58m 33s	**01h 10m 29s**	02h 04m 02s	01h 58m 22s
13	–	–	24h 25m 48s	**12h 31m 41s**
14	–	–	**01h 42m 59s**	02h 11m 38s
15	00h 20m 26s	00h 17m 05s	00h 21m 49s	**00h 15m 10s**
16	02h 14m 27s	02h 43m 55s	**00h 56m 47s**	00h 59m 37s
HF5	00h 00m 14s	00h 00m 23s	**00h 00m 08s**	00h 00m 11s
HF8	**00h 00m 31s**	00h 00m 42s	00h 00m 52s	00h 00m 58s
HF9	06h 16m 31s	01h 07m 23s	00h 14m 21s	**00h 09m 03s**

Eldarica; in fact, they are triggered in Eldarica's underlying SMT solver, Princess. We also exclude bugs 7 and 18. For these bugs, the solver returned an incorrect response because, as the developers informed us, the instances were beyond the scope of Golem and Ultimate Tree Automizer, respectively, e.g., by using a quantified constraint φ. Since extending the solvers to handle such instances is a major undertaking for the developers, they implemented a temporary fix by returning an error (instead of the incorrect response). Finally, we exclude bug 17, which was found in an abstract domain of Theta. The developers have currently disabled the domain and opened a new issue to revisit its soundness in the future.

We also measure how long it takes HornGator to detect fixed bugs found by HornFuzz [31]. To distinguish them from our bugs, we label the HornFuzz bugs using "HF" and a number; as usual, each label links to the corresponding bug report. HornFuzz detected a total of eleven fixed bugs; one of these turned out to be a duplicate, leaving us with ten. We exclude HF1 because it cannot be detected by HornGator (see RQ6 for more details). We also exclude HF2 because the fix was to disable the buggy solver option. Finally, we exclude HF3, HF4, HF6, HF7, and HF10 since no fuzzing is necessary to detect them. Specifically, the bugs are immediately detected by simply plugging in the returned model into certain existing CHC-Comp instances.

The first and last columns of Tab. 3 show the average time (across 20 fuzzing campaigns with different numeric random seeds) it takes HornGator to re-find the bugs discussed above. (The other columns are discussed in RQ5.) The average time to bug varies from 11s (for bug HF5 of severity 3a) to over 12h (for bug 13 of severity 1 and 2). Unsurprisingly, most of this time is spent on running the solver under test; in fact, for all bugs, this time is $\geq 85\%$, and for all but two bugs (8 and 13), this time is $\geq 95\%$.

RQ4: Transformations. When running the above experiment, we also inspected the sequences of transformations that led to revealing each bug. The transformations appearing most frequently in the sequences are AddCstrLHS (28%), UnplugLHS+Clause (14%), PlugTopLHS (14%), and FuseWeak (14%). Transformations DropRedundantClause (1%) and ReplaceClauseWithFact

(2%) are the least frequent since they may only be applied to Unsat instances for which a refutation proof is generated—refutation proofs cannot always be generated by certain solvers.

However, even these least frequent transformations were critical for bug detection. In particular, they appear in 3% and 9% of the sequences as the last transformation that revealed the bug. For example, bug 9 was only detected due to either DropRedundantClause or ReplaceClauseWithFact in all 20 runs.

All transformations were effective either by contributing new instances to the knowledge base that eventually led to a bug, or by being applied last and immediately triggering a bug.

Note that we do not claim that every transformation is strictly necessary for every bug. Computing a minimal set of transformations is computationally impractical: with 12 transformations, this would require exploring up to 2^{12} configurations. Moreover, removing a transformation that appears unnecessary for the currently known bugs might reduce diversity and prevent HornGator from uncovering other, as-yet-unknown bugs in future campaigns. Instead, RQ5 demonstrates that certain groups of transformations—particularly those leveraging models and proofs—are critical for HornGator's overall effectiveness.

RQ5: Interrogation vs. metamorphic testing. Interrogation testing extends metamorphic testing in two key ways: it leverages analysis witnesses and maintains a knowledge base. So, in addition to the solver response, HornGator uses generated witnesses—models or refutation proofs—to construct new CHC instances. It also integrates a knowledge base that accumulates diverse queries over time. To assess the effectiveness of these interrogation-testing extensions in the CHC setting, we introduce the following baselines.

HornGator w/o kb: This baseline has the same functionality as HornGator, but it does not use a knowledge base. Specifically, it always applies transformations to the instance that was generated last, and any previous instances are discarded. The FuseWeak and FuseStrong transformations cannot be applied as they require two instances as input.

HornGator w/o wtn: This baseline has the same functionality as HornGator, but it disables all transformations that use a witness, i.e., PlugModelLHS, PlugModelRHS, DropRedundantClause, and ReplaceClauseWithFact.

HornGator w/o kb&wtn: This baseline does not use a knowledge base and disables all transformations that use a witness; it is most similar to HornFuzz.

Note that the last two baselines are still able to detect bugs of severity 3a, where the model of a satisfiable instance is invalid. This is achieved by a model-validation step performed for all instances that are found Sat: it consists of plugging the entire model into the CHC instance—essentially converting it into a regular SMT instance—and checking its satisfiability.

The second, third, and fourth columns of Tab. 3 show the time to bug for these baselines. "–" indicates that none of the 20 fuzzing campaigns found the bug within 72h. The remaining bugs were found in all 20 campaigns, with the exception of bug 5, which was only found in 11 campaigns for HornGator w/o kb&wtn; the time is averaged across these 11 campaigns and is shown in italics in the table. For each bug, the lowest average time is shown in bold.

As shown in the table, HornGator is the only configuration that finds all bugs and is faster than the baselines for eight of them. In contrast, HornGator w/o kb&wtn fails to detect six bugs, four of which have severity 1, and only detects bug 5 in eleven out of 20 campaigns. All remaining bugs are found faster by HornGator with the exception of bugs 10 and HF8. For bug 10, the slowdown of 1.66x is related to the use of fusion transformations as described in the next paragraph. Bug HF8 is found in less than a minute by all configurations.

In comparison to HornGator, the baseline w/o kb fails to find three bugs, two of which have severity 1. For nine of the remaining bugs, HornGator achieves a speedup between 1.13x (bug 15)

and 13.87x (bug 6), whereas for the other four bugs, HornGator is slower between 1.27x (bug 8) and 7.78x (bug 10). The latter slowdown is significant, yet a clear outlier in the table. Upon investigation, we found that the slowdown is caused by the fusion transformations, which generate large instances that might take longer to solve. However, as we discussed in the previous RQ, these transformations are critical in detecting bugs; they constitute 24% of the transformations in all bug-revealing sequences and appear in 33% of the sequences as the last transformation that revealed the bug.

Finally, the baseline w/o wtn fails to find three bugs, two of which have severity 1. For eight of the remaining bugs, HornGator achieves a speedup between 1.03x (bug 2) and 5.75x (bug 5), whereas for the other five bugs, HornGator is slower between 1.05x (bug 16) and 1.67x (bug 11). Although small, such a slowdown may occur for bugs that do not require any witness transformations to be detected. In these cases, applying them simply increases the time to bug.

Overall, each extension of interrogation testing over metamorphic testing independently contributes to finding more bugs, and in most cases, faster. Using both constitutes the best configuration.

RQ6: Comparison to HornFuzz. HornFuzz uses metamorphic testing to test Spacer; as mentioned earlier, its design most closely resembles baseline HornGator w/o kb&wtn. In this RQ, we first discuss the fixed bugs found by HornFuzz. We then check if our bugs existed in the version of Spacer that HornFuzz tested. Last, we try to use HornFuzz to detect our bugs.

HornFuzz detected bugs HF1 and HF2 of severity 2. HF1 was found by a transformation that replaces a conjunct φ on the LHS of a clause with $\varphi \wedge \varphi$. By iteratively applying this transformation, HornFuzz built an instance with a clause containing over 1M conjuncts, which exceeded an internal limit of Spacer. Since HornGator does not duplicate conjuncts, it cannot trigger this bug. Even though the developers provided a fix, they said that this behavior is unlikely to be exposed in practice. HF2 uses a buggy option that was used for experiments a decade ago; the option is now removed.

HornFuzz also found bugs HF3, HF4, HF5, HF6, HF7, HF8, HF9, and HF10 of severity 3a. Recall that, even though HornFuzz does not use witnesses in its metamorphic transformations, it detects bugs of this severity with a model-validation check, i.e., by plugging the entire model into a satisfiable CHC instance. HornGator can detect all these bugs, typically in less than a minute. In our experiments, we noticed that five of them can be triggered by simply applying the model-validation check for existing satisfiable CHC-Comp instances; no fuzzing is necessary when using such instances as seeds. The remaining three bugs (HF5, HF8 and HF9) are among the ones with the shortest time to bug in Tab. 3.

Next, we discuss whether HornFuzz could have found our bugs but did not. To this end, we check whether they existed in Spacer in May'23, when HornFuzz was used to test the solver. HornGator detected seven Spacer bugs, namely 12, 13, 14, 15, 16, B, and C. Of these, four were already present in the old Spacer version and missed by HornFuzz, namely 12, 13, 14, and B.

Finally, we ran HornFuzz with the seed instances that revealed our 5 fixed Spacer bugs, namely 12, 13, 14, 15, and 16, and checked whether it can detect them. Bug 14 cannot be found by HornFuzz by construction as it relies on two CHC-Comp seeds. As for the remaining four bugs, HornFuzz could not detect them within the time limit of 72h that we used for our experiments.

Based on these findings, HornGator is significantly more effective in bug finding than HornFuzz.

5.3 Threats to Validity

The validity of our results depends on the CHC solvers under test, the seed instances, and the randomness in HornGator and its baselines. To limit the effect of these threats, we test five solvers that participate in CHC-Comp, use thousands of seed instances from CHC-Comp, and run our fuzzing campaigns with 20 numeric random seeds.

6 Related Work

HornFuzz. The most closely related work is HornFuzz [31], which proposes a fuzzing technique for CHC solvers based on metamorphic testing [12]. In contrast, HornGator is based on interrogation testing [18], an extension of metamorphic testing that was already used for testing program analyzers. It introduces two key innovations: (1) using more detailed information—such as witnesses—to design the transformations, and (2) using the knowledge base to form new (trick) queries by building on past results stored there. RQ6 compares the two fuzzers in terms of their bug-finding effectiveness.

SMT solvers. Several fuzzers have been proposed for SAT solvers (e.g., [8]), string solvers (e.g., [5, 9]), and SMT solvers (e.g., [6, 7, 20, 24, 29, 30, 34–38]). We refer the reader to Sect. 1 for a detailed discussion of why SMT fuzzers are not well suited for CHC solvers.

In brief, SMT fuzzers generate or mutate inputs tailored to the subset of SMT-LIB accepted by SMT solvers, which differs substantially from the structurally constrained input format required by CHC solvers. For example, Diver [20] applies unrestricted random mutations, and YinYang [36] introduces fresh free variables—both approaches that violate CHC syntax. Grammar-based generators such as ET [34] and Falcon [37] also miss CHC-specific constructs (e.g., uninterpreted predicates) and thus fail to trigger CHC-specific bugs.

SMT fuzzers also tend to focus on refutational soundness bugs—cases where a satisfiable input is incorrectly reported as Unsat—which are considered the most severe in the SMT setting. However, in the CHC domain, the most critical bugs (severity 1) occur when Sat is returned for an unsatisfiable instance. SMT fuzzers like Diver [20] and STORM [24], which only generate satisfiable inputs, are incapable of uncovering such errors.

Finally, SMT fuzzers that rely on differential testing, such as OpFuzz[35], TypeFuzz[30], Falcon [37], and ET [34], inherit the limitations discussed in Sect. 1, including theory mismatches, solver-specific constructs, and ambiguity when solvers disagree.

Datalog engines. CHC solvers are also connected to Datalog engines given that a Datalog constraint can be viewed as a specific type of Horn clause. Specifically, Datalog typically considers only predicates over finite domains instead of infinite domains, such as mathematical integers. As a result, CHC solvers are, in theory, more expressive than Datalog engines. There are three existing fuzzers for Datalog engines, namely, queryFuzz [23] and DLSmith [25], which are based on metamorphic testing, as well as Deopt [40], which is based on incremental rule evaluation.

Taking inspiration from our CHC transformations, Datalog engines also expose interesting additional information that could be used to design transformations for these engines. For instance, some Datalog engines expose provenance information to explain which rule applications were used to derive a given fact; this shares some similarity with refutation proofs, which explain how a solver derived a contradiction. None of the existing fuzzers rely on such detailed information.

Program analyzers. Beyond constraint solvers, there is a substantial body of work on testing more general-purpose program-analysis infrastructure—such as, abstract interpreters [10, 21, 28, 32], model checkers [21, 39], and symbolic execution engines [19]—and compilers [11].

7 Conclusion

We presented the first interrogation-testing technique for CHC solvers and implemented it in HornGator. Using models, refutation proofs, and a knowledge base, HornGator forms diverse CHC instances that are able to trigger bugs of severity 1–3. In fact, HornGator detected 21 unique bugs in five solvers, 18 of which have severity 1 or 2. We also showed that HornGator is significantly more effective in bug finding than related work.

In future work, we plan to further increase the effectiveness of HornGator by devising a diversity criterion that filters the instances that are added to the knowledge base, exploring how to better select instances to fuzz from the knowledge base (instead of doing it uniformly at random), and developing a power schedule that assigns more energy to "interesting" instances (i.e., fuzzes them more).

Data Availability

Our tool is publicly available at https://github.com/Rigorous-Software-Engineering/HornGator and archived at https://doi.org/10.5281/zenodo.19132854.

Acknowledgments

We thank the CHC-solver developers for their valuable work and assistance. This work was supported by Maria Christakis' ERC Starting grant 101076510.

References

[1] [n. d.]. CHC-COMP. https://chc-comp.github.io.
[2] [n. d.]. The Satisfiability Modulo Theories Library. http://smtlib.cs.uiowa.edu.
[3] Nikolaj S. Bjørner, Arie Gurfinkel, Kenneth L. McMillan, and Andrey Rybalchenko. 2015. Horn Clause Solvers for Program Verification. In *Fields of Logic and Computation II (LNCS, Vol. 9300)*. Springer, 24–51.
[4] Martin Blicha, Konstantin Britikov, and Natasha Sharygina. 2023. The Golem Horn Solver. In *CAV (LNCS, Vol. 13965)*. Springer, 209–223.
[5] Dmitry Blotsky, Federico Mora, Murphy Berzish, Yunhui Zheng, Ifaz Kabir, and Vijay Ganesh. 2018. StringFuzz: A Fuzzer for String Solvers. In *CAV (LNCS, Vol. 10982)*. Springer, 45–51.
[6] Mauro Bringolf, Dominik Winterer, and Zhendong Su. 2022. Finding and Understanding Incompleteness Bugs in SMT Solvers. In *ASE*. ACM, 43:1–43:10.
[7] Robert Brummayer and Armin Biere. 2009. Fuzzing and Delta-Debugging SMT Solvers. In *SMT*. ACM, 1–5.
[8] Robert Brummayer, Florian Lonsing, and Armin Biere. 2010. Automated Testing and Debugging of SAT and QBF Solvers. In *SAT (LNCS, Vol. 6175)*. Springer, 44–57.
[9] Alexandra Bugariu and Peter Müller. 2020. Automatically Testing String Solvers. In *ICSE*. ACM, 1459–1470.
[10] Alexandra Bugariu, Valentin Wüstholz, Maria Christakis, and Peter Müller. 2018. Automatically Testing Implementations of Numerical Abstract Domains. In *ASE*. ACM, 768–778.
[11] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. 2020. A Survey of Compiler Testing. *Comput. Surv.* 53 (2020), 4:1–4:36. Issue 1.
[12] Tsong Yueh Chen, S. C. Cheung, and Siu-Ming Yiu. 1998. *Metamorphic Testing: A New Approach for Generating Next Test Cases*. Technical Report HKUST–CS98–01. HKUST.
[13] Daniel Dietsch, Matthias Heizmann, Jochen Hoenicke, Alexander Nutz, and Andreas Podelski. 2019. Ultimate TreeAutomizer (CHC-COMP Tool Description). In *HCVS/PERR@ETAPS (EPTCS, Vol. 296)*. 42–47.
[14] Arie Gurfinkel and Nikolaj S. Bjørner. 2019. The Science, Art, and Magic of Constrained Horn Clauses. In *SYNASC*. IEEE Computer Society, 6–10.
[15] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. 2015. The SeaHorn Verification Framework. In *CAV (LNCS, Vol. 9206)*. Springer, 343–361.
[16] Hossein Hojjat and Philipp Rümmer. 2018. The ELDARICA Horn Solver. In *FMCAD*. IEEE Computer Society, 1–7.
[17] Temesghen Kahsai, Philipp Rümmer, Huascar Sanchez, and Martin Schäf. 2016. JayHorn: A Framework for Verifying Java Programs. In *CAV (LNCS, Vol. 9779)*. Springer, 352–358.
[18] David Kaindlstorfer, Anastasia Isychev, Valentin Wüstholz, and Maria Christakis. 2024. Interrogation Testing of Program Analyzers for Soundness and Precision Issues. In *ASE*. ACM, 319–330.
[19] Timotej Kapus and Cristian Cadar. 2017. Automatic Testing of Symbolic Execution Engines via Program Generation and Differential Testing. In *ASE*. IEEE Computer Society, 590–600.
[20] Jongwook Kim, Sunbeom So, and Hakjoo Oh. 2023. Diver: Oracle-Guided SMT Solver Testing with Unrestricted Random Mutations. In *ICSE*. IEEE Computer Society, 2224–2236.
[21] Christian Klinger, Maria Christakis, and Valentin Wüstholz. 2019. Differentially Testing Soundness and Precision of Program Analyzers. In *ISSTA*. ACM, 239–250.
[22] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. 2014. SMT-Based Model Checking for Recursive Programs. In *CAV (LNCS, Vol. 8559)*. Springer, 17–34.

[23] Muhammad Numair Mansur, Maria Christakis, and Valentin Wüstholz. 2021. Metamorphic Testing of Datalog Engines. In *ESEC/FSE*. ACM, 639–650.
[24] Muhammad Numair Mansur, Maria Christakis, Valentin Wüstholz, and Fuyuan Zhang. 2020. Detecting Critical Bugs in SMT Solvers Using Blackbox Mutational Fuzzing. In *ESEC/FSE*. ACM, 701–712.
[25] Muhammad Numair Mansur, Valentin Wüstholz, and Maria Christakis. 2023. Dependency-Aware Metamorphic Testing of Datalog Engines. In *ISSTA*. ACM, 236–247.
[26] Yusuke Matsushita, Takeshi Tsukada, and Naoki Kobayashi. 2020. RustHorn: CHC-Based Verification for Rust Programs. In *ESOP (LNCS, Vol. 12075)*. Springer, 484–514.
[27] William M. McKeeman. 1998. Differential Testing for Software. *Digital Technical Journal* 10 (1998), 100–107. Issue 1.
[28] Jan Midtgaard and Anders Møller. 2017. QuickChecking Static Analysis Properties. *Softw. Test., Verif. Reliab.* 27 (2017). Issue 6.
[29] Aina Niemetz, Mathias Preiner, and Armin Biere. 2017. Model-Based API Testing for SMT Solvers. In *SMT*. 10 pages.
[30] Jiwon Park, Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2021. Generative Type-Aware Mutation for Testing SMT Solvers. *PACMPL* 5 (2021), 1–19. Issue OOPSLA.
[31] Anzhela Sukhanova and Valentyn Sobol. 2023. HornFuzz: Fuzzing CHC solvers. In *EASE*. ACM, 83–92.
[32] Jubi Taneja, Zhengyang Liu, and John Regehr. 2020. Testing Static Analyses for Precision and Soundness. In *CGO*. ACM, 81–93.
[33] Tamás Tóth, Ákos Hajdu, András Vörös, Zoltán Micskei, and István Majzik. 2017. Theta: A Framework for Abstraction Refinement-Based Model Checking. In *FMCAD*. IEEE Computer Society, 176–179.
[34] Dominik Winterer and Zhendong Su. 2024. Validating SMT Solvers for Correctness and Performance via Grammar-Based Enumeration. *PACMPL* 8 (2024), 2378–2401. Issue OOPSLA2.
[35] Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2020. On the Unusual Effectiveness of Type-Aware Operator Mutations for Testing SMT Solvers. *PACMPL* 4 (2020), 193:1–193:25. Issue OOPSLA.
[36] Dominik Winterer, Chengyu Zhang, and Zhendong Su. 2020. Validating SMT Solvers via Semantic Fusion. In *PLDI*. ACM, 718–730.
[37] Peisen Yao, Heqing Huang, Wensheng Tang, Qingkai Shi, Rongxin Wu, and Charles Zhang. 2021. Fuzzing SMT Solvers via Two-Dimensional Input Space Exploration. In *ISSTA*. ACM, 322–335.
[38] Peisen Yao, Heqing Huang, Wensheng Tang, Qingkai Shi, Rongxin Wu, and Charles Zhang. 2021. Skeletal Approximation Enumeration for SMT Solver Testing. In *ESEC/FSE*. ACM, 1141–1153.
[39] Chengyu Zhang, Ting Su, Yichen Yan, Fuyuan Zhang, Geguang Pu, and Zhendong Su. 2019. Finding and Understanding Bugs in Software Model Checkers. In *ESEC/FSE*. ACM, 763–773.
[40] Chi Zhang, Linzhang Wang, and Manuel Rigger. 2024. Finding Cross-Rule Optimization Bugs in Datalog Engines. *PACMPL* 8 (2024), 110–136. Issue OOPSLA1.

Received 2025-09-10; accepted 2025-12-22