



Reasoning About Hyperproperties with Automated Theorem Provers (extended version)

Tobias Nießen, Ana Oliveira da Costa, Johannes Schoisswohl,
Thomas A. Henzinger and Laura Kovács

EasyChair preprints are intended for rapid
dissemination of research results and are
integrated with the rest of EasyChair.

September 27, 2026

Reasoning About Hyperproperties with Automated Theorem Provers (Extended Version)

Tobias Nießen¹, Ana Oliveira da Costa², Johannes Schoisswohl¹,
Thomas A. Henzinger², and Laura Kovács¹

¹ TU Wien, Austria

`{tobias.niessen,johannes.schoisswohl,laura.kovacs}@tuwien.ac.at`

² ISTA, Austria

`{tah,ana.costa}@ista.ac.at`

Abstract

We present a benchmark suite and an accompanying experimental study that assesses the capabilities of modern Automated Theorem Provers (ATPs) for hyperproperty reasoning. Existing tools for reasoning about hyperproperties typically rely on specialised model checkers or a custom encoding to satisfiability modulo theories (SMT) solvers or first-order provers. Our suite constitutes the first systematic collection of SMT-LIB encodings spanning hyperproperty satisfiability, model checking, and software verification via Constrained Horn Clauses (CHCs). Using this suite, we evaluate state-of-the-art ATPs, including SMT solvers and first-order theorem provers, that support both arithmetic and quantified reasoning. Beyond measuring how different solvers perform, we summarize lessons we learned on what makes a good encoding and solver configuration for hyperproperties. The benchmark suite, translation scripts, and experimental setup are released as an open artifact, enabling reproducible evaluation and providing a foundation for future improvements in automated reasoning about hyperproperties. This paper is the extended version of a paper with the same title which will be published at LPAR 2026.

1 Introduction

Hyperproperties [16] specify system requirements relating multiple system executions at once. They are commonly used in applications of system safety and security, for example, to capture non-interference and k -safety in program security [4, 27, 46], consistency in concurrent or distributed systems [29], or to quantify about system robustness [14]. Automating reasoning about hyperproperties is however challenging as it potentially requires handling arbitrary alternation of quantifiers over execution traces, while also embedding theory-specific proving steps, for example, within integer arithmetic. This combination of quantifiers and theories with arithmetic requires reasoning that remains one of the hardest challenges of automated theorem provers (ATPs), including satisfiability modulo theory (SMT) solvers and first-order theorem provers, as also evidenced by the yearly competitions in automated reasoning [41, 45].

In this paper, we study the challenge of hyperproperty reasoning in an experimental manner: we provide an extensive dataset for experimentation (Section 3) and evaluate automated theorem proving approaches using our problem collection (Section 4). Our work uses the expressivity of hypertrace logic [5, 13], allowing us to model hyperproperties in the first-order logic of uninterpreted functions with linear integer arithmetic (UFLIA). We consider three reasoning tasks related to hyperproperties, each task being concerned with a different aspect of the satisfiability/validity relation between a system and its hyperproperty specification, as follows:

- *Satisfiability*, where the functional behaviour is left unspecified: given a specification, we ask whether some system exists that satisfies it. A model is then a system whose executions satisfy the hyperproperty (Section 3.1).
- *(Bounded) Model Checking*: we ask whether a given system violates (a finite prefix of) a given hyperproperty specification (Section 3.2).
- *Verification via Constrained Horn Clauses*: we ask for a proof (in the form of an inductive invariant) that a system satisfies a hyperproperty specification (Section 3.3).

For each of these reasoning task that we consider, we construct an extensive benchmark set encoded in the SMT-LIB language [2] of automated reasoners. Our benchmark set currently consists of 1157 problem instances, collecting existing examples from hypertrace reasoning for satisfiability [24, 25] encoded in hypertrace logic, to model checking [17, 18, 31], and CHC solving [33, 34]. Our benchmark set is publicly available [39] and can be further extended, enabling experimentation and development in hyperproperty reasoning.

We evaluated our benchmark set on an extensive and diverse set of automated theorem provers, including SMT solvers, CHC solvers, and first-order theorem provers. We take a detailed look at the experimental results of the different benchmark groups in Section 4 and point out takeaways from our experimental evaluation in Section 5.

2 Hypertrace Logic and Hyperproperties

We model each execution of a system as a *trace* τ , formally defined as an infinite sequence of valuations of variables $\mathcal{X}$ over a domain $\mathcal{D}$ (i.e., $\tau \in (\mathcal{D}^{\mathcal{X}})^{\omega}$). A *system* is then a set $S \subseteq (\mathcal{D}^{\mathcal{X}})^{\omega}$ of traces. A *trace property* $T \subseteq (\mathcal{D}^{\mathcal{X}})^{\omega}$ defines a set of admissible traces, with the set of all trace properties being $\mathbb{T} = 2^{(\mathcal{D}^{\mathcal{X}})^{\omega}}$. We say that a *system* S *satisfies a trace property* T exactly when all its traces are in T (that is, $S \subseteq T$). A *hyperproperty* $\mathbf{T}$ is a set $\mathbf{T} \subseteq \mathbb{T}$ of admissible systems. A system S satisfies a hyperproperty $\mathbf{T}$ iff $S \in \mathbf{T}$. Satisfaction thus lifts from $S \subseteq T$ for trace properties to $S \in \mathbf{T}$ for hyperproperties.

We specify hyperproperties in a two-sorted first-order logic with a *time* sort $\mathbb{N}_{<}$, interpreted over $(\mathbb{N}, <)$, and an uninterpreted *trace* sort $\mathbb{T}$. Atoms relating the two sorts are binary uninterpreted predicates $X(\pi, i)$, which we read as “ X holds in trace π at time i .” Formally, our hyper formulas φ are defined by the grammar:

$$\varphi ::= \exists \pi :: \mathbf{S} \varphi \mid \exists i \varphi \mid \neg \varphi \mid \varphi \vee \varphi \mid i < j \mid i = j \mid X(\pi, i) \quad (1)$$

where π is a trace variable from a set $\mathcal{V}_{\mathbb{T}}$, i and j are time variables from a set $\mathcal{V}_{\mathbb{N}}$ disjoint from $\mathcal{V}_{\mathbb{T}}$, and X ranges over a finite set $\mathbb{X}$ of binary predicates. We may use lowercase letters for the predicates of $\mathbb{X}$ to reflect their correspondence with propositional variables in traces. The guarded quantifier $\exists \pi :: \mathbf{S} \varphi$ restricts π to the traces satisfying a unary predicate $\mathbf{S}$ (typically the executions of a given system, with different guards referring to different systems) and corresponds, in plain FOL, to $\exists \pi (\mathbf{S}(\pi) \wedge \varphi)$; dually, $\forall \pi :: \mathbf{S} \varphi$ corresponds to $\forall \pi (\mathbf{S}(\pi) \rightarrow \varphi)$. We use the usual abbreviations $\forall x \varphi \stackrel{\text{def}}{=} \neg(\exists x \neg \varphi)$, with $x \in \{i, \pi\}$, and $\varphi \wedge \varphi' \stackrel{\text{def}}{=} \neg(\neg \varphi \vee \neg \varphi')$. We refer to this logic as *hypertrace logic* [5, 13], which lies in the first-order logic of uninterpreted functions with linear integer arithmetic (UFLIA): uninterpreted predicates over the trace sort, combined with the order over time, definable in linear integer arithmetic. Since UFLIA provides uninterpreted functions, we can also accommodate traces over non-Boolean domains, reading each $X(\pi, i)$ as a function producing the value of X on π at i and taking atoms to be comparisons between

$$\begin{aligned}
\text{OD}_{\text{gl}} &\stackrel{\text{def}}{=} \forall \pi :: S \forall \pi' :: S (\forall i \geq 0 \text{ in}(\pi, i) = \text{in}(\pi', i)) \rightarrow (\forall i \geq 0 \text{ out}(\pi, i) = \text{out}(\pi', i)) \\
\text{OD}_{\text{init}} &\stackrel{\text{def}}{=} \forall \pi :: S \forall \pi' :: S (\text{in}(\pi, 0) = \text{in}(\pi', 0) \rightarrow (\forall i \geq 0 \text{ out}(\pi, i) = \text{out}(\pi', i))) \\
\text{QNI}(n) &\stackrel{\text{def}}{=} \forall \pi_0 :: S. \dots \forall \pi_n :: S \neg (\bigwedge_{i \leq n} \text{in}(\pi_0, 0) = \text{in}(\pi_i, 0) \wedge \bigwedge_{i \neq j} \text{out}(\pi_i, 0) \neq \text{out}(\pi_j, 0)) \\
\text{QNI}^t(n) &\stackrel{\text{def}}{=} \forall \pi_0 :: S. \dots \forall \pi_n :: S \neg (\bigwedge_{i \leq n} (\forall k \geq 0 \text{ in}(\pi_0, k) = \text{in}(\pi_i, k)) \wedge \bigwedge_{i < j} (\forall k \geq 0 \text{ out}(\pi_i, k) \neq \text{out}(\pi_j, k))) \\
\text{OCS}(k) &\stackrel{\text{def}}{=} \forall \pi :: S (\text{secret}(\pi, k) \rightarrow (\exists \pi' :: S (\text{public}(\pi', k) \wedge (\forall 0 \leq i \leq k \text{ obs}(\pi, i) = \text{obs}(\pi', i))))) \\
\text{OKS}(k) &\stackrel{\text{def}}{=} \forall n \geq 0 \forall \pi :: S (\text{secret}(\pi, n) \rightarrow (\exists \pi' :: S (\text{public}(\pi', n) \wedge (\forall 0 \leq i \leq k+n \text{ obs}(\pi, i) = \text{obs}(\pi', i)))))
\end{aligned}$$

Figure 1: Representative hyperproperties in our **satisfiability** set. The OD variants differ in how inputs are compared; QNI scales the number of trace quantifiers with n , shown with/without temporal operators; the opacity formulas OCS and OKS alternate trace/time quantifiers.

such terms; we adopt this generalization in our benchmarks. The mix between trace and time quantifiers makes the problem of verifying hyperproperties challenging for automated theorem provers, as discussed in Section 3.

3 Benchmark Suite for Hyperproperty Reasoning

We construct a new benchmark set for hyperproperty reasoning, by using the SMT-LIB language standard [2]. We organize our benchmarks within three categories, reflecting the source and usage of their respective hyperproperty queries. Our **satisfiability** set (Section 3.1) is hand-crafted: it encodes implications between hyperproperties, expressed in hypertrace logic, directly into first-order logic. The **model-checking** (Section 3.2) and **chc-verification** (Section 3.3) sets are extracted from existing tools and case studies in the literature. The former consists of SMT queries with explicit quantification over base sorts (e.g., `Bool`, `Int`); the latter captures finding inductive invariants via Constrained Horn Clauses (CHCs).

3.1 Hyperproperty Satisfiability

The satisfiability problem has as input only the specification: given a hyperproperty, we ask whether there exists a system that satisfies it. Such satisfiability queries are useful when a requirement admits several distinct formalizations and one is interested in determining whether they are consistent or how they relate. We refer to this set of benchmarks as **satisfiability**. Here we are, in particular, interested in checking *implications between different hyperproperties*; that is, given two formulas, φ_A and φ_B , we check whether $\varphi_A \Rightarrow \varphi_B$ is valid. We present a direct encoding of this problem into first-order logic, with the goal of producing benchmarks that can be solved by, and used to evaluate, ATP solvers. Concretely, each instance asserts the premise φ_A as an axiom together with the negated conclusion $\neg \varphi_B$ (marked by the `:goal` annotation). Satisfiability has previously been studied in the context of HyperLTL [8, 23, 24] (the fragment of hypertrace logic where all trace quantifiers precede any time quantification [13]), via reductions to LTL satisfiability, quantified boolean formulas (QBF), and first-order logic (through an intermediate automaton construction). Our encodings go beyond HyperLTL, keeping our benchmarks independent of domain-specific formalizations.

The **satisfiability** benchmark set checks implications between variants of several hyperproperties, each chosen to probe a different source of difficulty; see Fig. 1 for representatives. *Observational determinism* (`od_smtlib`) [46] and *symmetry in mutual-exclusion protocols* (`sym_smtlib`) [24] are classical examples of hyperproperties. For each of them we collect

several formalizations from the literature [8, 23, 24] to check the implications among them. This group provides a baseline (due to its simpler quantification pattern) while also illustrating how solvers can clarify the relationships between subtly different variants. For instance, OD_{init} and OD_{gl} in Fig. 1 differ only in whether executions must agree on the whole input sequence or just the initial input. With the set of benchmarks for *quantitative noninterference* (QNI) we want to stress-test the number of trace quantifiers, which grows with the leakage bound n . We include a time-free version $\text{QNI}(n)$ (`qni_smtlib`) and a temporal version $\text{QNI}^t(n)$ (`qni_t_smtlib`) to separate the difficulty of quantification from that of reasoning over time (i.e., over integers). We also include a third set on *opacity* variants (`opacity_smtlib`) [37] contributing specifications with varied alternation patterns between trace and time quantifiers, ranging from *current-state k -opacity* (OCS in Fig. 1), where the trace quantifiers precede the time quantifier, as in the earlier formulas, to *k -step opacity* (OKS in Fig. 1) that begins with a time quantifier, then trace quantifiers, and closes with another time quantifier.

3.2 Model Checking Hyperproperties

A common use case of ATPs in the context of hyperproperties arises from model checking and verification of hyperproperty specifications against concrete or abstract systems. We contribute the **model-checking** benchmark set as a collection of SMT encodings used by model checking and verification algorithms for hyperproperties.

Bounded model checkers directly benefit from the efficiency of modern SAT/SMT solvers. When a query is satisfiable and proves the existence of a counterexample to the expected property, a SAT solver typically finds such a satisfying assignment quickly. However, up until that point, every other query will be unsatisfiable, and CDCL-style SAT/SMT solvers take much longer to show that a query is unsatisfiable than to find a satisfying assignment [38]. In other words, bounded model checkers typically spend the vast majority of their solving time on unsatisfiable instances, and then increase their bound until the resulting query becomes satisfiable or until all system behavior has been explored [9, 15, 17, 18]. We include in our **model-checking** benchmark set SMT encodings of both satisfiable and unsatisfiable queries, and use the following tools as the basis of obtaining our SMT encodings.

$\text{HY}\exists\text{N}\forall$ [17, 18] was the first SMT-based bounded model checking tool for the class of $\forall\exists$ hyperproperties. Because $\text{HY}\exists\text{N}\forall$ processes $\forall^+\exists^+$ hyperproperties and negates them during this instantiation phase while also reducing multiple consecutive quantifiers of the same kind to a single quantifier, it issues $\exists\forall$ -quantified SMT queries to the solver. We modify $\text{HY}\exists\text{N}\forall$ to more closely match traditional bounded model checking approaches; in particular, we use an SMT encoding of all paths of a certain (observational) length simultaneously, whereas the original tool considered each path independently [9, Sec. 10.4.2].

HYPERQB 2.0 [31] encodes instantiations of hyperproperty specifications against finite-state systems as SMT queries. Unlike $\text{HY}\exists\text{N}\forall$, HYPERQB represents the system through a state-dependent concurrent transition relation [9, Sec. 10.3 and 10.4.1]. By default, HYPERQB 2.0 [31] (here, labeled *Default SMT mode*) does not negate the specification. For example, for a $\forall\exists$ -quantified hyperproperty, while $\text{HY}\exists\text{N}\forall$ [18] would issue $\exists\forall$ -quantified SMT queries in an attempt to find a counterexample, HYPERQB 2.0 issues $\forall\exists$ -quantified queries with no free variables to show the absence of a counterexample. This is possible because HYPERQB 2.0 uses the Z3 SMT solver [19], which supports almost arbitrary quantifier alternation, whereas $\text{HY}\exists\text{N}\forall$ uses Yices' specialized exists-forall solver [22, 26, 43]. Nevertheless, HYPERQB 2.0 similarly produces $\exists\forall$ -quantified SMT queries in its witness-producing mode, in which case the free variables of the query can be translated into a counterexample. We have labeled our

benchmarks derived in this mode as *Witness-producing mode*.

Our **model-checking** benchmark set uses and extends the benchmarks of $\text{HY}\exists\text{N}\forall$ and HYPERQB 2.0. In the case of HYPERQB 2.0, this means a collection of benchmarks with specifications expressed in a variant of HyperLTL and finite-state transition. For $\text{HY}\exists\text{N}\forall$, benchmarks are drawn from multiple sources and encoded for $\text{HY}\exists\text{N}\forall$'s input format, which uses a fragment of OHyperLTL [6] for specifications and an imperative programming language for specifying the program under test. The groups *HY $\exists\text{N}\forall$ examples* and *HY $\exists\text{N}\forall$ benchmarks* are from [17, 18]. The group *ORHLE benchmarks* refers to (a large subset of) those benchmarks that were used for the evaluation of ORHLE, the software implementation accompanying RHLE [20]. These benchmarks are non-reactive, i.e., specifications refer to initial inputs and final outputs only. Not all functions are guaranteed to terminate, making these benchmarks inherently asynchronous, and the domain of values is infinite in many cases. The group *hypo benchmarks (fixed observational depth)* is based on benchmarks that were used for the evaluation of HYPA [6] and that represent asynchronous reactive systems over infinite value domains, about half of which fall into the $\forall^+$ fragment of the hyperlogic OHyperLTL.

3.3 Hyperproperty Verification via Constrained Horn Clauses

Recent works rely on Constrained Horn Clauses (CHCs) for the verification of relational properties against concrete systems [33]. CHCs are a fragment of first-order logic that extends Horn clauses with constraints over a background theory (e.g., linear arithmetic, arrays, or algebraic datatypes). Verification of regular trace properties using CHC encodings is well-established [1, 11, 28], and has further motivated the development of efficient CHC solvers [10, 12, 30, 36, 44].

This approach has recently been extended to hyperproperties (with support for quantifier alternation), and the promising nature and increasing popularity of this use of CHCs makes it an important target for the purpose of our evaluation. Our **chc-verification** benchmark set consists of CHC instances (in SMT-LIB format) that we obtain from a modified version of HYHORN [33], which reduces the verification of $\forall^*\exists^*$ hyperproperties to solving CHC constraint systems using Z3/Spacer [36]. Following the original evaluation of HYHORN [33], we categorize our benchmarks by whether or not a user-supplied predicate abstraction is used to aid the solver (labeled *with predicate abstraction/without predicate abstraction*) and by whether or not the benchmark solves *full instances* as opposed to *simplified instances*. HYHORN's evaluation draws its benchmarks from multiple sources, which we categorize and label as follows.

- *HYPA's k -safety benchmarks* is the set of k -safety benchmarks that only universally quantify over k -tuples of executions used for the evaluation of HYPA [6].
- *HYPA's benchmarks beyond k -safety* is the set of $\forall\exists$ -quantified hyperproperty benchmarks used for the evaluation of HYPA [6].
- *PDSC comparator benchmarks* is the set of benchmarks used for the evaluation of Property Directed Self Composition (PDSC) [40] for comparators, which correspond to k -safety hyperproperties [34, Appendix A].
- *More k -safety benchmarks* contains the k -safety benchmarks listed in [33, Table 1] except those that are in *HYPA's k -safety benchmarks* or *PDSC comparator benchmarks*.

Our **chc-verification** benchmark set also includes 55 instances of a parameterized benchmark (*HY $\exists\text{N}\forall$'s escalating benchmarks*) that was originally used to compare $\text{HY}\exists\text{N}\forall$ [17, 18] to (an only slightly modified version of) HYHORN [33] as well as other tools. Notably, the runtime of the tools $\text{HY}\exists\text{N}\forall$ [18], HYPERQB [32], HYHORN [33], and AutoHyper [7] has been shown to grow rapidly as the parameter increases [18, Sec. 5.4]. We remark that these previous evaluations used version 4.12.2 of the SMT solver Z3 for solving CHC queries [18, 33].

4 Experimental Evaluation

We are interested in evaluating weaknesses, strengths and performance of state-of-the-art ATPs on our benchmark set. For that we selected the most efficient ATPs from the annual competitions on SMT solving, CHC solving, and first-order theorem proving. Namely, we used all solvers submitted to [SMTCOMP-2025](#) that competed in the [Single Query Track of the Arith Division](#) with error score 0; the leading three solvers of each division of [CHCCOMP-2025](#) which supplied pre-built binaries on their websites; and the VAMPIRE theorem prover [3] which won all divisions of [CASC-30](#) in 2025. As detailed later, we also included additional versions of the solvers YICES [21], Z3 [19], and VAMPIRE. A full list of solvers used is given in Fig. 2.

We ran each solver on each problem of our benchmark set using a single core of an AMD EPYC 7252 8-core processor with two threads per core and 125GiB of RAM. A soft timeout of 120 seconds was given, and the processes were killed after a hard timeout of 132 seconds.

Fig. 2 gives an overview of the results of these runs.¹ In the benchmark category **satisfiability**, 396 of 449 problems were solved; in **model-checking** 193 of 203, whereas in **chc-verification** 445 of 505 benchmarks were solved. For some benchmarks there were disagreements between the results reported by solvers; in all of these cases, one solver reported **sat/unsat** while all other solvers agreed on the opposite, thus we assume that the deviating solver’s result was unsound. Such a scenario happened, once for LOAT, and 6 times for SMTITP. This suggests that our benchmark set is structurally different from usual competition benchmarks, as it reveals previously unknown bugs in these solvers.

Experimental Analysis using Competition Solvers. We detail experimental findings for the competition solvers. These are all but the dashed or dotted lines in the plots Fig. 2. The first striking fact is that Z3-ALPHA outperforms all other solvers in every category in **sat** and **unsat** problems. Secondly, as not all solvers support all reasoning features of our benchmark set, some solvers were not able to solve any benchmarks in some categories. For example, SMTRAT could not solve any problems at all as it can only be used with (non-linear) real arithmetic, but not with integers or uninterpreted functions. LOAT is only designed for solving linear constrained horn clauses, thus it was not applicable for our benchmark set **chc-verification**, and only applicable to 8 out of 203 **model-checking** problems. VAMPIRE and IPROVER are both first-order theorem provers, thus designed for showing unsatisfiability which is reflected in them not solving any **sat** instances.

In **chc-verification**, GOLEM [12] is faster and solves more problems than the other CHC solvers; yet, still behind the SMT solver Z3. We also see that UNIHORN starts solving with a significant delay. The curve of UNIHORN suggests that more problems can be solved if the timeout is increased. Nevertheless, due to the logarithmic time scale, thus we expect that not many more problems can be solved within a practically useful timeout. Detailed results¹ show that surprisingly GOLEM solved the same number of **chc-verification** problems with or without predicate abstraction.

Within **satisfiability**, we observe that many solvers are able to solve **unsat** problems, but only Z3 and SMTITP are able to establish **sat**. As these problems use many quantifiers, our results suggest that there is a lack of methods for establishing **sat** in a quantified context, compared to techniques for **unsat** problems.

Insider Knowledge in Experimental Setup. In our experiments, we found different versions of some solvers exhibiting unexpected results for some benchmarks. As explained in Section 3.2, some benchmarks in **model-checking** were generated using HYEN $\forall$ [18], which uses version 2.6.4 of YICES [22, 21] as the solver for its SMT queries. Investigating our benchmark

¹Table 1 in Section B details the results for each category of our benchmark set.

Solver	Competition	Version
 Z3-ALPHA	SMTCOMP-2025	SMTCOMP Submission z3alpha
 CVC5	SMTCOMP-2025	SMTCOMP Submission cvc5
 iPROVER	SMTCOMP-2025	SMTCOMP Submission iprover
 SMTITP	SMTCOMP-2025	SMTCOMP Submission smtinterpol
 SMTRAT	SMTCOMP-2025	SMTCOMP Submission smtrat
 YICESQS	SMTCOMP-2025	SMTCOMP Submission yicesQS
 ELDARICA	CHCCOMP-2025	2.3 (latest release)
 GOLEM	CHCCOMP-2025	0.9.0 (latest release)
 LoAT	CHCCOMP-2025	0.9.8 (latest release)
 UNIHORN	CHCCOMP-2025	0.3.1 (latest release)
 VAMPIRE	CASC-30	5.0.1 (latest release)
 Z3-4.12.2	—	4.12.2
 YICES-2.6.4	—	2.6.4
 VAMPIRE _g	—	Commit 94b98f5 ; Portfolio with goal annotations [39]
 VAMPIRE _g	—	Commit 94b98f5 ; Portfolio ignoring goal annotations [39]

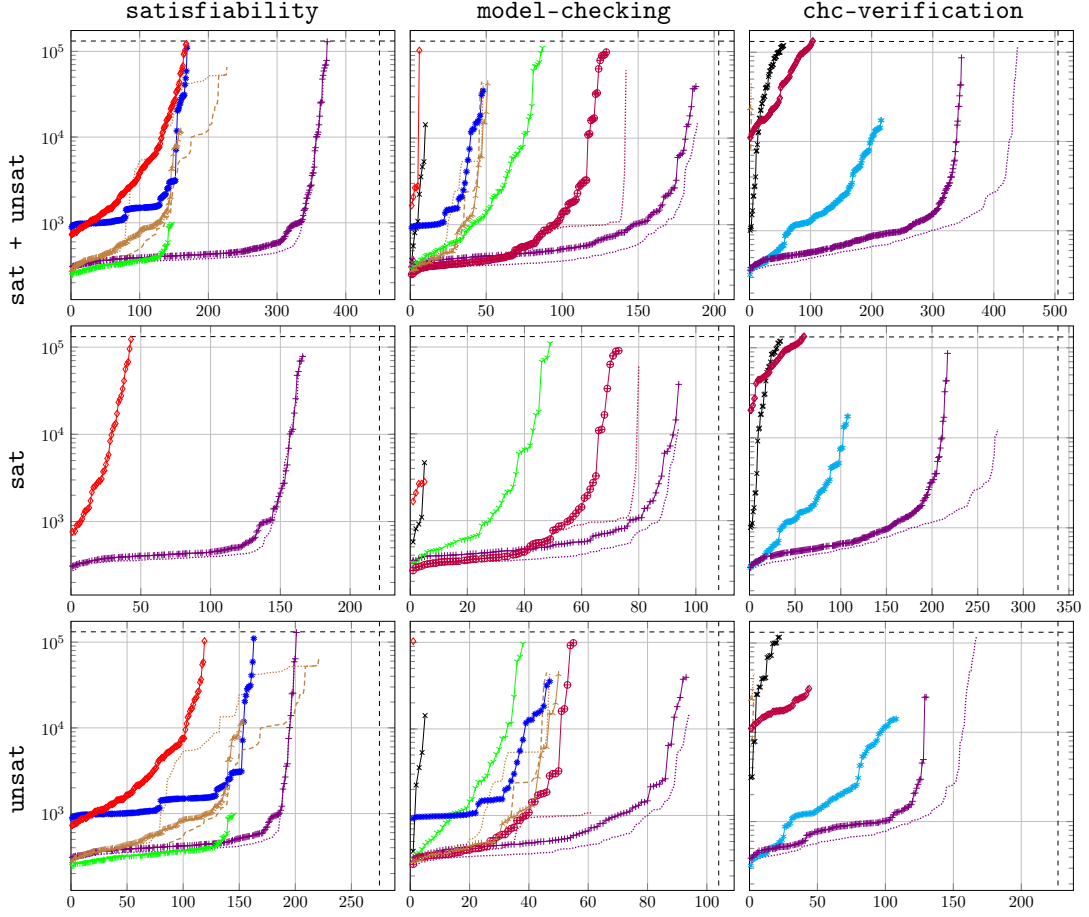


Figure 2: Cumulative number of our benchmarks problems solved by each solver. The y axis shows the time in ms in logarithmic scale. The x axis shows the number of problems solved. A dot in the graph means that a solver solved x problems with a timeout of y ms per problem. The first row shows all solved problems, the second row only problems shown to be **sat**, and the last one only **unsat** ones. The dashed lines hard timeout, and the total number of benchmarks in the respective category. This includes problems with unknown **sat/unsat** status.

results, we found that the SMTCOMP submission YICESQS did not solve a number of benchmarks, which were known to be solved by YICES in HY $\exists\forall$. This is due to the fact that YICESQS is a custom solver that extends YICES in the sense that it uses YICES as a backend, but with custom algorithms for handling quantifiers, whereas HY $\exists\forall$ uses YICES’ built-in algorithms for quantifiers [22]. We included both YICES 2.6.4 and YICESQS in our evaluation. Fig. 2 shows that the plain version YICES-2.6.4 vastly outperforms YICESQS.

As mentioned in Section 3.3, many of HY $\exists\forall$ ’s **escalating** benchmarks from the **chc-verification** benchmark set were observed to be solvable in [18] with rapidly growing runtime depending on the benchmark parameter. Nevertheless, none of the competition solvers we considered were able to solve any of the problems in our experiments. As HYHORN [33] used Z3-4.12.2 as its solver [35], we also include this older version of Z3 in our experiments. Indeed, this version solves HY $\exists\forall$ ’s **escalating** benchmarks with the expected runtime behaviour², and also performs much better for the PDSC comparator benchmarks without predicate abstraction. Fig. 2 shows that Z3-4.12.2 overall outperforms Z3-ALPHA.

We initially experimented with TPTP [42] encodings of the benchmark group **satisfiability**. Surprisingly, VAMPIRE exhibited different runtime behaviour depending on which of the (equivalent) encodings was used. This happened since firstly VAMPIRE chose a different portfolio for TPTP problems than the one used for SMT-LIB, and secondly, in TPTP conjecture formulas are annotated, which allows VAMPIRE to use goal-oriented strategies. Using these insights we created new portfolios for VAMPIRE. VAMPIRE can be supplied with a sampler file, which specifies a random distribution of option combinations. Using such a sampler file, we generated a new portfolio as follows. First, we create a random 50/50 split of the problem set into a training set (P_{train}) and test set (P_{test}). We started with $P = P_{\text{train}}$, chose a timeout within 1 and 10 seconds, created 80 random option combinations, and ran each of these across P . We chose the best configuration, added it to the portfolio, removed all solved problems from P , and repeated this process for 24 hours with 64 threads in parallel. VAMPIRE using this portfolio is referred to as VAMPIRE* in Fig. 2. Samplers and portfolios are publicly available [39].

We also evaluate how much VAMPIRE’s goal-oriented strategies improve its reasoning capacities. We extended VAMPIRE to parse `:goal` annotations in SMT-LIB syntax, which are standard-conform but do not carry any specific semantics for other solvers. For every problem of the shape $\varphi_A \rightarrow \varphi_B$ in **satisfiability**, we annotated φ_B as goal. We then created another portfolio, with the same method as above, but additionally adding the option to enable `:goal` parsing to the sampler file. We refer to the resulting solver as VAMPIRE_g* in Fig. 2. Fig. 2 evaluates the portfolios on both P_{test} and P_{train} . Overall results³ for P_{test} only follow similar trends. In **satisfiability**, the new portfolios significantly outperform VAMPIRE, and when it comes to **unsat** problems they slightly outperform Z3-4.12.2. In our other benchmark categories, barely any difference in performance compared to VAMPIRE can be seen, indicating that in VAMPIRE’s options are not sufficient for solving more problems in **model-checking** and **chc-verification**, while its proof search for the quantified problems in **satisfiability** is a matter of finding the right set of options rather. We see hardly any difference in performance between VAMPIRE_g* and VAMPIRE* in the numbers of solved problems; the timings in **satisfiability** shows that VAMPIRE_g* solves problems faster.

²This can be seen from Fig. 4 in Section B.

³These results can be found in Fig. 3, Section B.

5 Conclusions

We created a diverse set of benchmarks for hyperproperty reasoning and evaluated state-of-the-art automated theorem provers on these examples, including SMT solvers, CHC solvers, and first-order theorem provers. Our benchmarks cover several relevant and complex reasoning tasks, ranging from the satisfiability of hyperproperty specifications to bounded model checking and to the verification of a given hyperproperty specification against a concrete system.

As many solvers have a vast number of options, they use portfolio approaches to select a useful subset of option combinations. As showcased by building custom portfolios for VAMPIRE, choosing the right combination of options can significantly improve performance. Thus, when choosing a solver, it is important to investigate the proof search options available and select from them carefully, or to use a portfolio approach. Annotations that enable goal-oriented proof search also help to improve performance.

Acknowledgements. This project was partially funded by the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation programme (grant agreement No. 101034440); by the ERC Proof of Concept Grant LEARN 101213411; by the Austrian Science Fund through the SFB project SpyCoDe 10.55776/F85 and its subprojects F8502 and F8504; by the Vienna Science and Technology Fund (WWTF) through [ForSmart Grant ID: 10.47379/ICT22007]; and by the Austrian Research Promotion Agency (FFG) through the COMET K1 SBA.

References

- [1] Emanuele De Angelis, Fabio Fioravanti, John P. Gallagher, Manuel V. Hermenegildo, Alberto Pettorossi, and Maurizio Proietti. Analysis and transformation of constrained horn clauses for program verification. *Theory and Practice of Logic Programming*, 22(6):974–1042, 2022. doi:10.1017/S1471068421000211.
- [2] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org, 2016.
- [3] Filip Bártěk, Ahmed Bhayat, Robin Coutelier, Márton Hajdú, Matthias Hetzenberger, Petra Hozzová, Laura Kovács, Jakob Rath, Michael Rawson, Giles Reger, Martin Suda, Johannes Schoisswohl, and Andrei Voronkov. The Vampire Diary. In Ruzica Piskac and Zvonimir Rakamaric, editors, *Computer Aided Verification - 37th International Conference, CAV 2025, Zagreb, Croatia, July 23-25, 2025, Proceedings, Part III*, volume 15933 of *Lecture Notes in Computer Science*, pages 57–71. Springer, 2025. doi:10.1007/978-3-031-98682-6_4.
- [4] Gilles Barthe, Renate Eilers, Pamina Georgiou, Bernhard Gleiss, Laura Kovács, and Matteo Maffei. Verifying relational properties using trace logic. In Clark W. Barrett and Jin Yang, editors, *2019 Formal Methods in Computer Aided Design, FMCAD 2019, San Jose, CA, USA, October 22-25, 2019*, pages 170–178. IEEE, 2019. doi:10.23919/FMCAD.2019.8894277.
- [5] Ezio Bartocci, Thomas Ferrère, Thomas A. Henzinger, Dejan Nickovic, and Ana Oliveira da Costa. Flavors of sequential information flow. In Bernd Finkbeiner and Thomas Wies, editors, *Verification, Model Checking, and Abstract Interpretation - 23rd International Conference, VMCAI 2022, Philadelphia, PA, USA, January 16-18, 2022, Proceedings*, volume 13182 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2022. doi:10.1007/978-3-030-94583-1_1.

- [6] Raven Beutner and Bernd Finkbeiner. Software verification of hyperproperties beyond k-safety. In Sharon Shoham and Yakir Vizel, editors, *Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part I*, volume 13371 of *Lecture Notes in Computer Science*, pages 341–362, Cham, 2022. Springer. doi:10.1007/978-3-031-13185-1_17.
- [7] Raven Beutner and Bernd Finkbeiner. Autohyper: Explicit-state model checking for HyperLTL. In Sriram Sankaranarayanan and Natasha Sharygina, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2023, Paris, France, April 22-27, 2023, Proceedings, Part I*, volume 13993 of *Lecture Notes in Computer Science*, pages 145–163, Cham, 2023. Springer. doi:10.1007/978-3-031-30823-9_8.
- [8] Raven Beutner and Bernd Finkbeiner. Checking satisfiability of hyperproperties using first-order logic. In S. Akshay, Aina Niemetz, and Sriram Sankaranarayanan, editors, *Automated Technology for Verification and Analysis - 22nd International Symposium, ATVA 2024, Kyoto, Japan, October 21-25, 2024, Proceedings, Part II*, volume 15055 of *Lecture Notes in Computer Science*, pages 198–211, Cham, 2024. Springer. doi:10.1007/978-3-031-78750-8_10.
- [9] Armin Biere and Daniel Kröning. SAT-based model checking. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 277–303. Springer, 2018. doi:10.1007/978-3-319-10575-8_10.
- [10] Nikolaj Bjørner, Ken McMillan, and Andrey Rybalchenko. On solving universally quantified horn clauses. In Francesco Logozzo and Manuel Fähndrich, editors, *Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings*, volume 7935 of *Lecture Notes in Computer Science*, pages 105–125, Berlin, Heidelberg, 2013. Springer. doi:10.1007/978-3-642-38856-9_8.
- [11] Nikolaj S. Bjørner, Arie Gurfinkel, Kenneth L. McMillan, and Andrey Rybalchenko. Horn clause solvers for program verification. In Lev D. Beklemishev, Andreas Blass, Nachum Dershowitz, Bernd Finkbeiner, and Wolfram Schulte, editors, *Fields of Logic and Computation II - Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday*, volume 9300 of *Lecture Notes in Computer Science*, pages 24–51, Cham, 2015. Springer. doi:10.1007/978-3-319-23534-9_2.
- [12] Martin Blicha, Konstantin Britikov, and Natasha Sharygina. The Golem horn solver. In Constantin Enea and Akash Lal, editors, *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part II*, volume 13965 of *Lecture Notes in Computer Science*, pages 209–223, Cham, 2023. Springer. doi:10.1007/978-3-031-37703-7_10.
- [13] Marek Chalupa, Thomas A. Henzinger, and Ana Oliveira da Costa. Flavors of quantifiers in hyperlogics. In C. Aiswarya, Ruta Mehta, and Subhajit Roy, editors, *45th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2025, BITS Pilani, K K Birla Goa Campus, India, December 17-19, 2025*, volume 360 of *LIPICs*, pages 20:1–20:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2025. doi:10.4230/LIPICs.FSTTCS.2025.20.
- [14] Swarat Chaudhuri, Sumit Gulwani, and Roberto Lubliner. Continuity and robustness of programs. *Commun. ACM*, 55(8):107–115, 2012. doi:10.1145/2240236.2240262.
- [15] Edmund M. Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001. doi:10.1023/a:1011276507260.
- [16] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium, CSF 2008, Pittsburgh, Pennsylvania, USA, 23-25 June 2008*, pages 51–65. IEEE Computer Society, 2008. doi:10.1109/CSF.2008.7.
- [17] Arthur Correnson, Tobias Nießen, Bernd Finkbeiner, and Georg Weissenbacher. Finding $\forall\exists$ hyperbugs using symbolic execution. *Proc. ACM Program. Lang.*, 8(OOPSLA2):1420–1445, Oct 2024. doi:10.1145/3689761.

- [18] Arthur Correnson, Tobias Nießen, Bernd Finkbeiner, and Georg Weissenbacher. Symbolic execution for refuting $\forall\exists$ hyperproperties. *Acta Informatica*, 62(4):40, Oct 2025. doi:10.1007/s00236-025-00504-z.
- [19] Leonardo de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340, Berlin, Heidelberg, 2008. Springer. doi:10.1007/978-3-540-78800-3_24.
- [20] Robert Dickerson, Qianchuan Ye, Michael K. Zhang, and Benjamin Delaware. RHLE: Modular Deductive Verification of Relational $\forall\exists$ Properties. In Ilya Sergey, editor, *Programming Languages and Systems - 20th Asian Symposium, APLAS 2022, Auckland, New Zealand, December 5, 2022, Proceedings*, volume 13658 of *Lecture Notes in Computer Science*, pages 67–87, Cham, 2022. Springer. doi:10.1007/978-3-031-21037-2_4.
- [21] Bruno Dutertre. Yices 2.2. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 737–744. Springer, July 2014. doi:10.1007/978-3-319-08867-9_49.
- [22] Bruno Dutertre. Solving exists/forall problems with Yices. In *13th International Workshop on Satisfiability Modulo Theories (SMT 2015)*, July 2015.
- [23] Bernd Finkbeiner, Christopher Hahn, and Tobias Hans. MGHYper: Checking satisfiability of hyperltl formulas beyond the $\exists^*\forall^*$ fragment. In Shuvendu K. Lahiri and Chao Wang, editors, *Automated Technology for Verification and Analysis - 16th International Symposium, ATVA 2018, Los Angeles, CA, USA, October 7-10, 2018, Proceedings*, volume 11138 of *Lecture Notes in Computer Science*, pages 521–527. Springer, 2018. doi:10.1007/978-3-030-01090-4_31.
- [24] Bernd Finkbeiner, Christopher Hahn, and Marvin Stenger. EAHYper: Satisfiability, implication, and equivalence checking of hyperproperties. In Rupak Majumdar and Viktor Kuncak, editors, *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*, volume 10427 of *Lecture Notes in Computer Science*, pages 564–570, Cham, 2017. Springer. doi:10.1007/978-3-319-63390-9_29.
- [25] Bernd Finkbeiner, Christopher Hahn, and Hazem Torfah. Model checking quantitative hyperproperties. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*, volume 10981 of *Lecture Notes in Computer Science*, pages 144–163, Cham, 2018. Springer. doi:10.1007/978-3-319-96145-3_8.
- [26] Adrià Gascón, Pramod Subramanian, Bruno Dutertre, Ashish Tiwari, Dejan Jovanović, and Sharad Malik. Template-based circuit understanding. In *Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, October 21-24, 2014*, pages 83–90. IEEE, October 2014. doi:10.1109/FMCAD.2014.6987599.
- [27] Joseph A. Goguen and José Meseguer. Security policies and security models. In *1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 26-28, 1982*, pages 11–20. IEEE Computer Society, 1982. doi:10.1109/SP.1982.10014.
- [28] Arie Gurfinkel. Program verification with constrained horn clauses (invited paper). In Sharon Shoham and Yakir Vizel, editors, *Computer Aided Verification - 34th International Conference, CAV 2022, Haifa, Israel, August 7-10, 2022, Proceedings, Part I*, volume 13371 of *Lecture Notes in Computer Science*, pages 19–29, Cham, 2022. Springer. doi:10.1007/978-3-031-13185-1_2.
- [29] Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990. doi:10.1145/78969.78972.

- [30] Hossein Hojjat and Philipp Rümmer. The ELDARICA horn solver. In Nikolaj S. Bjørner and Arie Gurfinkel, editors, *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018*, pages 1–7. IEEE, 2018. doi:10.23919/FMCAD.2018.8603013.
- [31] Tzu-Han Hsu, Milad Rabizadeh, Kenneth Rogale, Fedor Filippov, Marco A. de Oliveira Batista, César Sánchez, and Borzoo Bonakdarpour. HyperQB: A bounded model checker for hyperproperties. *CoRR*, abs/2109.12989, 2025. doi:10.48550/arXiv.2109.12989.
- [32] Tzu-Han Hsu, César Sánchez, and Borzoo Bonakdarpour. Bounded model checking for hyperproperties. In Jan Friso Groote and Kim Guldstrand Larsen, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part I*, volume 12651 of *Lecture Notes in Computer Science*, pages 94–112, Cham, 2021. Springer. doi:10.1007/978-3-030-72016-2_6.
- [33] Shachar Itzhaky, Sharon Shoham, and Yakir Vizel. Hyperproperty verification as CHC satisfiability. In Stephanie Weirich, editor, *Programming Languages and Systems - 33rd European Symposium on Programming, ESOP 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part II*, volume 14577 of *Lecture Notes in Computer Science*, pages 212–241, Cham, 2024. Springer. doi:10.1007/978-3-031-57267-8_9.
- [34] Shachar Itzhaky, Sharon Shoham, and Yakir Vizel. Hyperproperty verification as CHC satisfiability. *CoRR*, abs/2304.12588, 2024. doi:10.48550/arXiv.2304.12588.
- [35] HyHorn ESOP’24 artifact evaluation, Jan 2024. doi:10.5281/zenodo.10449234.
- [36] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. SMT-based model checking for recursive programs. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 17–34, Cham, 2014. Springer. doi:10.1007/978-3-319-08867-9_2.
- [37] Siyuan Liu, Ashutosh Trivedi, Xiang Yin, and Majid Zamani. Secure-by-construction synthesis of cyber-physical systems. *Annu. Rev. Control.*, 53:30–50, 2022. doi:10.1016/j.arcontrol.2022.03.004.
- [38] João Marques-Silva, Inês Lynce, and Sharad Malik. *Conflict-Driven Clause Learning SAT Solvers*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, chapter 4, pages 133–182. IOS Press, Feb 2021. doi:10.3233/FAIA200987.
- [39] Tobias Niefen, Ana Oliveira da Costa, Johannes Schoisswohl, Thomas A. Henzinger, and Laura Kovács. Dataset for reasoning about hyperproperties with automated theorem provers. Zenodo, September 2026. doi:10.5281/zenodo.22801767.
- [40] Ron Shemer, Arie Gurfinkel, Sharon Shoham, and Yakir Vizel. Property directed self composition. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I*, volume 11561 of *Lecture Notes in Computer Science*, pages 161–179, Cham, 2019. Springer. doi:10.1007/978-3-030-25540-4_9.
- [41] Geoff Sutcliffe. The CADE ATP system competition - CASC. *AI Magazine*, 37(2):99–101, 2016. doi:10.1609/aimag.v37i2.2620.
- [42] Geoff Sutcliffe. Stepping stones in the TPTP world. In Christoph Benzmüller, Marijn J. H. Heule, and Renate A. Schmidt, editors, *Automated Reasoning - 12th International Joint Conference, IJCAR 2024, Nancy, France, July 3-6, 2024, Proceedings, Part I*, number 14739 in *Lecture Notes in Artificial Intelligence*, pages 30–50. Springer, 2024. doi:10.1007/978-3-031-63498-7_3.
- [43] Ashish Tiwari, Adrià Gascón, and Bruno Dutertre. Program synthesis using dual interpretation. In Amy P. Felty and Aart Middeldorp, editors, *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*, volume 9195 of *Lecture Notes in Computer Science*, pages 482–497. Springer, August 2015. doi:10.1007/978-3-319-21401-6_33.

- [44] Tamás Tóth, Ákos Hajdu, András Vörös, Zoltán Micskei, and István Majzik. Theta: A framework for abstraction refinement-based model checking. In Daryl Stewart and Georg Weissenbacher, editors, *2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017*, pages 176–179. IEEE, 2017. doi:[10.23919/FMCAD.2017.8102257](https://doi.org/10.23919/FMCAD.2017.8102257).
- [45] Tjark Weber, Sylvain Conchon, David Déharbe, Matthias Heizmann, Aina Niemetz, and Giles Reger. The SMT competition 2015-2018. *J. Satisf. Boolean Model. Comput.*, 11(1):221–259, 2019. doi:[10.3233/SAT190123](https://doi.org/10.3233/SAT190123).
- [46] Steve Zdancewic and Andrew C. Myers. Observational determinism for concurrent program security. In *16th IEEE Computer Security Foundations Workshop (CSFW-16 2003), 30 June - 2 July 2003, Pacific Grove, CA, USA*, page 29. IEEE Computer Society, 2003. doi:[10.1109/CSFW.2003.1212703](https://doi.org/10.1109/CSFW.2003.1212703).

A Description of Sources of Benchmarks

A.1 Satisfiability

Observational Determinism (OD) requires that a system's public observations be deterministic in its inputs: any two executions that agree on public inputs must also agree on public outputs, so that no private data can influence what is publicly visible. What it means to *agree on inputs and outputs*, however, is not uniquely defined: different choices of grouping and temporal scope lead to subtly different specifications. We encode four formalizations of OD:

- In the global view, two executions with the same input sequence must produce the same output sequence:

$$\text{OD}_{\text{gl}} \stackrel{\text{def}}{=} \forall \pi :: S \forall \pi' :: S (\forall i \geq 0 \text{ in}(\pi, i) = \text{in}(\pi', i)) \rightarrow (\forall i \geq 0 \text{ out}(\pi, i) = \text{out}(\pi', i)) \quad (2)$$

- In the initial-input view, two executions with the same initial input must produce the same output sequence:

$$\text{OD}_{\text{init}} \stackrel{\text{def}}{=} \forall \pi :: S \forall \pi' :: S (\text{in}(\pi, 0) = \text{in}(\pi', 0) \rightarrow (\forall i \geq 0 \text{ out}(\pi, i) = \text{out}(\pi', i))) \quad (3)$$

- In the output view, two executions must produce the same outputs up to the first step at which their inputs differ:

$$\begin{aligned} \text{OD}_{\text{out}} \stackrel{\text{def}}{=} \forall \pi :: S \forall \pi' :: S (\forall k \geq 0 \text{ out}(\pi, k) = \text{out}(\pi', k)) \\ \vee (\exists i \geq 0 (\text{in}(\pi, i) \neq \text{in}(\pi', i) \wedge \forall 0 \leq j < i \text{ out}(\pi, j) = \text{out}(\pi', j))) \end{aligned} \quad (4)$$

- In the step-wise view, equal inputs at a step force equal outputs at that step:

$$\text{OD}_{\text{step}} \stackrel{\text{def}}{=} \forall \pi :: S \forall \pi' :: S \forall i \geq 0 (\text{in}(\pi, i) = \text{in}(\pi', i) \rightarrow \text{out}(\pi, i) = \text{out}(\pi', i)) \quad (5)$$

The formulas OD_{gl} , OD_{init} , and OD_{out} are from [24], where the authors prove the implications $\text{OD}_{\text{init}} \Rightarrow \text{OD}_{\text{gl}}$, $\text{OD}_{\text{init}} \Rightarrow \text{OD}_{\text{out}}$, and $\text{OD}_{\text{out}} \Rightarrow \text{OD}_{\text{gl}}$. In our benchmarks, we also include specifications restricted to classes of systems. Concretely:

- *Input-constant systems* (CI) are those whose input never changes over time:

$$\text{CI} \stackrel{\text{def}}{=} \forall \pi :: S \forall i \geq 0 \text{ in}(\pi, i) = \text{in}(\pi, i + 1) \quad (6)$$

- *Input-enabled systems* (IE) realize every input behavior: the universal trace ranges over *all* traces (the unconstrained quantification of hypertrace logic [13]), and the system S (the range of the existential) must contain an execution with the same input:

$$\text{IE} \stackrel{\text{def}}{=} \forall \pi \exists \pi' :: S \forall i \geq 0 \text{ in}(\pi, i) = \text{in}(\pi', i) \quad (7)$$

To check an implication over one of these classes, we restrict the trace quantifiers of both OD specifications to the system S (i.e., we guard them by S) and add the class property as an extra assumption. For instance, over input-constant systems we check whether CI together with one S -restricted specification entails another.

Quantitative Noninterference (QNI) relaxes OD by allowing a bounded amount of leakage: for any given observable input, the system may exhibit at most n executions with pairwise different observable outputs. Following [24], we compare QNI specifications for different leakage bounds (in our benchmarks we compare them up to $n = 12$):

$$\text{QNI}(n) \stackrel{\text{def}}{=} \forall \pi_0 :: \mathbf{S} \dots \forall \pi_n :: \mathbf{S} \neg \left(\bigwedge_{i \leq n} \text{in}(\pi_0, 0) = \text{in}(\pi_i, 0) \wedge \bigwedge_{i \neq j} \text{out}(\pi_i, 0) \neq \text{out}(\pi_j, 0) \right) \quad (8)$$

This definition uses no temporal operators; it compares inputs and outputs only at the initial step. Adapting [25], we also consider a temporal variant that lifts the comparison to whole input and output sequences:

$$\begin{aligned} \text{QNI}^t(n) \stackrel{\text{def}}{=} \forall \pi_0 :: \mathbf{S} \dots \forall \pi_n :: \mathbf{S} \neg \left(\bigwedge_{i \leq n} (\forall k \geq 0 \text{ in}(\pi_0, k) = \text{in}(\pi_i, k)) \wedge \right. \\ \left. \bigwedge_{i < j} (\forall k \geq 0 \text{ out}(\pi_i, k) \neq \text{out}(\pi_j, k)) \right) \end{aligned} \quad (9)$$

Like $\text{QNI}(n)$, this asserts that no $n + 1$ traces can agree on inputs at every step while their outputs remain pairwise distinct at every step.

Symmetry in Mutual-Exclusion Protocols requires that no process has an inherent advantage over the others; equivalently, the protocol's behavior is independent of process identities. Following [24], we compare several specifications of symmetry between two processes. The two-process specifications [24] are:

$$\begin{aligned} \text{Sym}_1 \stackrel{\text{def}}{=} \forall \pi_1 :: \mathbf{S} \forall \pi'_1 :: \mathbf{S} (\forall i \geq 0 (\text{select}(\pi_1, i) \neq \text{select}(\pi'_1, i) \wedge \text{pause}(\pi_1, i) = \text{pause}(\pi'_1, i))) \\ \rightarrow (\forall i \geq 0 (\text{pc1}(\pi_1, i) = \text{pc2}(\pi'_1, i) \wedge \text{pc2}(\pi_1, i) = \text{pc1}(\pi'_1, i))) \end{aligned} \quad (10)$$

$$\begin{aligned} \text{Sym}_2 \stackrel{\text{def}}{=} \forall \pi_2 :: \mathbf{S} \forall \pi'_2 :: \mathbf{S} (\forall i \geq 0 (\text{select}(\pi_2, i) \neq \text{select}(\pi'_2, i) \wedge \text{pause}(\pi_2, i) = \text{pause}(\pi'_2, i)) \\ \wedge \text{select}(\pi_2, i) < 3 \wedge \text{select}(\pi'_2, i) < 3)) \\ \rightarrow (\forall i \geq 0 (\text{pc1}(\pi_2, i) = \text{pc2}(\pi'_2, i) \wedge \text{pc2}(\pi_2, i) = \text{pc1}(\pi'_2, i))) \end{aligned} \quad (11)$$

$$\begin{aligned} \text{Sym}_3 \stackrel{\text{def}}{=} \forall \pi_3 :: \mathbf{S} \forall \pi'_3 :: \mathbf{S} (\forall i \geq 0 (\text{select}(\pi_3, i) \neq \text{select}(\pi'_3, i) \wedge \text{pause}(\pi_3, i) = \text{pause}(\pi'_3, i)) \\ \wedge \text{select}(\pi_3, i) < 3 \wedge \text{select}(\pi'_3, i) < 3 \wedge \text{symbreak}(\pi_3, i) \neq \text{symbreak}(\pi'_3, i))) \\ \rightarrow (\forall i \geq 0 (\text{pc1}(\pi_3, i) = \text{pc2}(\pi'_3, i) \wedge \text{pc2}(\pi_3, i) = \text{pc1}(\pi'_3, i))) \end{aligned} \quad (12)$$

$$\begin{aligned} \text{Sym}_4 \stackrel{\text{def}}{=} \forall \pi_4 :: \mathbf{S} \forall \pi'_4 :: \mathbf{S} (\forall k \geq 0 (\text{pc1}(\pi_4, k) = \text{pc2}(\pi'_4, k) \wedge \text{pc2}(\pi_4, k) = \text{pc1}(\pi'_4, k))) \\ \vee (\exists i \geq 0 (\neg (\text{select}(\pi_4, i) \neq \text{select}(\pi'_4, i) \wedge \text{pause}(\pi_4, i) = \text{pause}(\pi'_4, i)) \\ \wedge \forall 0 \leq j < i (\text{pc1}(\pi_4, j) = \text{pc2}(\pi'_4, j) \wedge \text{pc2}(\pi_4, j) = \text{pc1}(\pi'_4, j)))))) \end{aligned} \quad (13)$$

Sym_4 is the safety weakening of Sym_1 , obtained by dropping its liveness requirement: for some systems it is equivalent, and easier, to verify that the matching condition holds until the symmetry assumption is first violated, rather than the implication between the two global conditions.

Opacity is a confidentiality property defined over a *secret* predicate marking the confidential runs of a system [37]. An intruder who knows the system's dynamics and observes its outputs tries to infer whether a run is secret; the system is *opaque* when it always retains plausible deniability (i.e., every secret run has a public run with the same observations, so the intruder can never be certain a secret occurred). We include several variants that differ in which secret is hidden and over which observation window:

- Current-state k -opacity (OCS(k)) hides a secret at step k over observations $[0, k]$:

$$\text{OCS}(k) \stackrel{\text{def}}{=} \forall \pi :: \mathbf{S} \left(\text{secret}(\pi, k) \rightarrow (\exists \pi' :: \mathbf{S} (\text{public}(\pi', k) \wedge (\forall 0 \leq i \leq k \text{ obs}(\pi, i) = \text{obs}(\pi', i)))) \right) \quad (14)$$

- k -step opacity (OKS(k)) hides a secret at any step n over $[0, k + n]$:

$$\text{OKS}(k) \stackrel{\text{def}}{=} \forall n \geq 0 \forall \pi :: \mathbf{S} \left(\text{secret}(\pi, n) \rightarrow (\exists \pi' :: \mathbf{S} (\text{public}(\pi', n) \wedge (\forall 0 \leq i \leq k+n \text{ obs}(\pi, i) = \text{obs}(\pi', i)))) \right) \quad (15)$$

- Pre-opacity (OPre(n)) ensures that observing the first n steps never commits to a future secret, i.e., for every horizon m , a public execution at step $n+m$ matches those observations:

$$\text{OPre}(n) \stackrel{\text{def}}{=} \forall m \geq 0 \forall \pi :: \mathbf{S} \exists \pi' :: \mathbf{S} \left(\text{public}(\pi', n+m) \wedge (\forall 0 \leq i \leq n \text{ obs}(\pi, i) = \text{obs}(\pi', i)) \right) \quad (16)$$

A.2 Model checking

We briefly describe the structure of SMT queries obtained from (a modified version of) HY $\exists\mathbf{N}\forall$ [18] and from HYPERQB 2.0 [31].

A.2.1 SMT queries obtained from HY $\exists\mathbf{N}\forall$

The model checking tool HY $\exists\mathbf{N}\forall$ uses an SMT solver (namely, YICES [21]) to find counterexamples to a given $\forall^+\exists^+$ hyperproperty given a concrete program under test. HY $\exists\mathbf{N}\forall$ further reduces multiple consecutive quantifiers of the same kind to a single quantifier.

Paths through the control flow graph of the program are encoded symbolically, and their length is limited by some bound k . On such a path P of bounded length k , only a finite number v_P of arithmetic values are computed, and whether or not an execution follows this particular path is a boolean condition C_P on these arithmetic values.

Let $\mathcal{V}$ be the domain of the value of each program variable, and let $\mathcal{V}_P = \mathcal{V}^{v_P}$ be the composite domain of the v_P many arithmetic values computed along path P . Having fixed a hyperproperty specification $\forall^P \pi_1. \exists^Q \pi_2. \varphi(\pi_1, \pi_2)$, where P and Q are programs, as well as some finite bound k , let $\mathcal{P}_k^P$ and $\mathcal{P}_k^Q$ be the sets of symbolic paths of length k within the control flow graph of program P and program Q , respectively. Further, given a pair of paths $P_1 \in \mathcal{P}_k^P$, $P_2 \in \mathcal{P}_k^Q$, let $\varphi_k^{P_1 \times P_2}$ be obtained by instantiating the specification body φ over the arithmetic values computed along paths P_1 and P_2 .

Then, HY $\exists\mathbf{N}\forall$ produces queries of the following form, modulo simplifications implemented within HY $\exists\mathbf{N}\forall$:

$$\bigvee_{P_1 \in \mathcal{P}_k^P} \exists x \in \mathcal{V}_{P_1}. (C_{P_1}(x) \wedge \bigwedge_{P_2 \in \mathcal{P}_k^Q} \forall y \in \mathcal{V}_{P_2}. (C_{P_2}(y) \implies \neg \varphi_k(x, y))).$$

This formula is satisfiable if there is some path $P_1 \in \mathcal{P}_k^P$ and some execution along this path within the program P such that no trace exists in program Q that satisfies the relation defined by φ . Because $\text{HY}\exists\text{N}\forall$'s benchmarks generally rely on linear integer arithmetic, this formula – when encoded in SMT-LIB syntax – is a query within the **LIA** logic. In some cases, $\text{HY}\exists\text{N}\forall$'s internal optimizations eliminate one of the quantifiers, in which case $\text{HY}\exists\text{N}\forall$ can often issue a simpler **QF_LIA** query instead.

We refer readers to [17, 18] for a detailed description of the translation to first-order formulas.

A.2.2 SMT queries obtained from **HYPERQB 2.0**

HYPERQB 2.0 does not consume control flow graphs in the same fashion that $\text{HY}\exists\text{N}\forall$ does; instead, it parses NuSMV or Verilog files and translates these into symbolic representations of an equivalent transition system [31].

HYPERQB 2.0, by default, does not negate the hyperproperty specification during this translation, which means that a generated query is satisfiable if the hyperproperty holds, and unsatisfiable otherwise. For example, for a $\forall^+\exists^+$ hyperproperty, this results in $\forall^+\exists^+$ -quantified SMT queries, which the SMT solver **Z3** [19] that is used by **HYPERQB 2.0** indeed supports. Due to this design decision, in the latter case, **HYPERQB 2.0** does not produce counterexamples by default. In its witness-producing mode, the tool does negate the property, and the free variables of the query can then be translated into a counterexample. We include SMT queries generated in either mode in our benchmark set.

We refer readers to [31, 32] for a detailed description of the generated queries.

B Supplementary Material for Experimental Evaluation

Benchmarks (#)	Z3-4.12.2	Z3-ALPHA	VAMPIRE*	VAMPIRE*	CVC5	I/PROVER	GOLEM	VAMPIRE	SMT/TP	YICES-QS	UNHORN	ELDARICA	LOAT	SAT/RAT	solved
all (1157)	1001 (69)	908 (3)	278 (0)	278 (1)	234 (2)	217 (0)	216 (0)	213 (0)	174 (0)	142 (1)	104 (0)	66 (2)	2 (0)	0 (0)	1034
model-checking (203)	189 (1)	188 (0)	48 (0)	47 (0)	87 (2)	48 (0)	0 (0)	51 (0)	6 (0)	129 (0)	0 (0)	10 (0)	2 (0)	0 (0)	193
Benchmarks based on HyperQB 2.0 (103)	90 (1)	89 (0)	1 (0)	1 (0)	87 (2)	0 (0)	0 (0)	0 (0)	0 (0)	84 (0)	0 (0)	9 (0)	0 (0)	0 (0)	93
A-BTL benchmarks (21)	15 (1)	14 (0)	0 (0)	0 (0)	13 (1)	0 (0)	0 (0)	0 (0)	0 (0)	14 (0)	0 (0)	0 (0)	0 (0)	0 (0)	16
HyperLTL benchmarks (Part 1) (30)	23 (0)	23 (0)	0 (0)	0 (0)	24 (1)	0 (0)	0 (0)	0 (0)	0 (0)	24 (0)	0 (0)	0 (0)	0 (0)	0 (0)	25
Default SMT mode (18)	14 (0)	14 (0)	0 (0)	0 (0)	15 (1)	0 (0)	0 (0)	0 (0)	0 (0)	15 (0)	0 (0)	0 (0)	0 (0)	0 (0)	16
Witness-producing mode (12)	9 (0)	9 (0)	0 (0)	0 (0)	9 (0)	0 (0)	0 (0)	0 (0)	0 (0)	9 (0)	0 (0)	0 (0)	0 (0)	0 (0)	9
HyperLTL benchmarks (Part 2) (46)	46 (0)	46 (0)	1 (0)	1 (0)	46 (0)	0 (0)	0 (0)	0 (0)	0 (0)	46 (0)	0 (0)	0 (0)	0 (0)	0 (0)	46
Default SMT mode (29)	29 (0)	29 (0)	1 (0)	1 (0)	29 (0)	0 (0)	0 (0)	0 (0)	0 (0)	29 (0)	0 (0)	0 (0)	0 (0)	0 (0)	29
Witness-producing mode (17)	17 (0)	17 (0)	0 (0)	0 (0)	17 (0)	0 (0)	0 (0)	0 (0)	0 (0)	17 (0)	0 (0)	0 (0)	0 (0)	0 (0)	17
Hardware model checking benchmarks (Verilog) (6)	6 (0)	6 (0)	0 (0)	0 (0)	4 (0)	0 (0)	0 (0)	0 (0)	0 (0)	6 (0)	0 (0)	0 (0)	0 (0)	0 (0)	6
Benchmarks based on HY3SV (100)	99 (0)	99 (0)	47 (0)	46 (0)	0 (0)	48 (0)	0 (0)	50 (0)	0 (0)	100 (1)	0 (0)	0 (0)	2 (0)	0 (0)	100
HY3SV examples (30)	30 (0)	30 (0)	12 (0)	11 (0)	0 (0)	12 (0)	0 (0)	12 (0)	0 (0)	30 (0)	0 (0)	0 (0)	1 (0)	0 (0)	30
bypa benchmarks (fixed observational depth) (13)	13 (0)	13 (0)	13 (0)	12 (0)	0 (0)	12 (0)	0 (0)	13 (0)	0 (0)	13 (0)	0 (0)	0 (0)	1 (0)	0 (0)	13
ORHLE benchmarks (27)	27 (0)	27 (0)	12 (0)	12 (0)	0 (0)	12 (0)	0 (0)	13 (0)	0 (0)	27 (0)	0 (0)	0 (0)	0 (0)	0 (0)	27
api-refinement (7)	7 (0)	7 (0)	4 (0)	4 (0)	0 (0)	4 (0)	0 (0)	4 (0)	0 (0)	7 (0)	0 (0)	0 (0)	0 (0)	0 (0)	7
blackjack (3)	3 (0)	3 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	3 (0)	0 (0)	0 (0)	0 (0)	0 (0)	3
delayed-release (9)	9 (0)	9 (0)	4 (0)	4 (0)	0 (0)	4 (0)	0 (0)	4 (0)	0 (0)	9 (0)	0 (0)	0 (0)	0 (0)	0 (0)	9
flaky-tests (1)	1 (0)	1 (0)	1 (0)	1 (0)	0 (0)	1 (0)	0 (0)	1 (0)	0 (0)	1 (0)	0 (0)	0 (0)	0 (0)	0 (0)	1
gpi (7)	7 (0)	7 (0)	3 (0)	3 (0)	0 (0)	3 (0)	0 (0)	3 (0)	0 (0)	7 (0)	0 (0)	0 (0)	0 (0)	0 (0)	7
HY3SV benchmarks (30)	29 (0)	29 (0)	10 (0)	11 (0)	0 (0)	12 (0)	0 (0)	12 (0)	0 (0)	30 (1)	0 (0)	0 (0)	0 (0)	0 (0)	30
satisfiability (449)	373 (2)	373 (1)	226 (0)	227 (1)	147 (0)	169 (0)	0 (0)	160 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	386
ci.od.smtlib (16)	8 (0)	8 (0)	13 (0)	14 (1)	7 (0)	8 (0)	0 (0)	8 (0)	2 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	14
is.od.bool.smtlib (16)	8 (0)	8 (0)	8 (0)	8 (0)	7 (0)	8 (0)	0 (0)	8 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	8
is.od.smtlib (16)	8 (0)	8 (0)	8 (0)	8 (0)	7 (0)	8 (0)	0 (0)	8 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	8
od.smtlib (16)	15 (0)	15 (0)	8 (0)	8 (0)	5 (0)	8 (0)	0 (0)	8 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	15
opacity.smtlib (81)	73 (2)	72 (1)	24 (0)	24 (0)	21 (0)	24 (0)	0 (0)	24 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	74
api.smtlib (144)	104 (0)	105 (0)	78 (0)	78 (0)	78 (0)	78 (0)	0 (0)	68 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	120
api.t.smtlib (144)	141 (0)	141 (0)	78 (0)	77 (0)	12 (0)	25 (0)	0 (0)	26 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	141
sym.smtlib (16)	16 (0)	16 (0)	9 (0)	10 (0)	10 (0)	10 (0)	0 (0)	10 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	16
che-verification (507)	439 (66)	317 (2)	4 (0)	4 (0)	0 (0)	0 (0)	0 (0)	216 (0)	0 (0)	0 (0)	0 (0)	56 (2)	0 (0)	0 (0)	445
with predicate abstraction (225)	222 (0)	223 (0)	2 (0)	2 (0)	0 (0)	0 (0)	0 (0)	108 (0)	0 (0)	0 (0)	0 (0)	37 (1)	0 (0)	0 (0)	224
full instances (126)	123 (0)	124 (0)	1 (0)	1 (0)	0 (0)	0 (0)	0 (0)	54 (0)	0 (0)	0 (0)	0 (0)	23 (1)	0 (0)	0 (0)	125
HyPA's benchmarks beyond k -safety (16)	14 (0)	15 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	4 (0)	0 (0)	0 (0)	15
HyPA's k -safety benchmarks (11)	10 (0)	10 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	5 (1)	0 (0)	0 (0)	11
More k -safety benchmarks (96)	3 (0)	3 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	3
PDSC comparator benchmarks (3)	96 (0)	96 (0)	1 (0)	1 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	96
simplified instances (90)	99 (0)	99 (0)	1 (0)	1 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	99
HyPA's k -safety benchmarks (3)	3 (0)	3 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	3
PDSC comparator benchmarks (96)	96 (0)	96 (0)	1 (0)	1 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	96
without predicate abstraction (280)	217 (66)	124 (2)	2 (0)	2 (0)	0 (0)	0 (0)	0 (0)	108 (0)	0 (0)	0 (0)	0 (0)	14 (0)	0 (0)	0 (0)	221
full instances (181)	121 (40)	69 (2)	1 (0)	1 (0)	0 (0)	0 (0)	0 (0)	54 (0)	0 (0)	0 (0)	0 (0)	11 (1)	0 (0)	0 (0)	125
HY3SV's accumulating benchmarks (55)	13 (13)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	13
HyPA's benchmarks beyond k -safety (16)	9 (1)	10 (1)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	11
HyPA's k -safety benchmarks (11)	3 (0)	4 (1)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	2 (1)	0 (0)	0 (0)	5
More k -safety benchmarks (9)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0
PDSC comparator benchmarks (96)	96 (26)	55 (0)	1 (0)	1 (0)	0 (0)	0 (0)	0 (0)	54 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	96
simplified instances (90)	96 (26)	55 (0)	1 (0)	1 (0)	0 (0)	0 (0)	0 (0)	54 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	96
HyPA's k -safety benchmarks (3)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0
PDSC comparator benchmarks (96)	96 (26)	55 (0)	1 (0)	1 (0)	0 (0)	0 (0)	0 (0)	54 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	96

Table 1: Results of running each solver on each problem group. The first column “ n (c)” gives the name n of the problem group, and the number of problems c in that group. Each cell “ s (u)” in the remaining columns lists that n problems where solved by a given solver in the respective benchmark group, and that u problems where uniquely solved by this solver and no other solver.

Benchmarks (#)	Z3-4.12.2	Z3-ALPHA	CVC5	VAMPIRE*	VAMPIRE†	VAMPIRE‡	VAMPIRE§	IPROVER	VAMPIRE	SMTTP	YICESQS	YICES-2.6.4	UNHORN	ELDARICA	LOAT	SMT-RAT	solved
all (579)	504 (34)	453 (1)	130 (1)	125 (0)	123 (0)	104 (0)	102 (0)	98 (0)	89 (0)	81 (0)	79 (0)	49 (0)	27 (0)	1 (0)	0 (0)	112	
model-checking (119)	111 (0)	111 (0)	56 (1)	21 (0)	19 (0)	0 (0)	21 (0)	23 (0)	5 (0)	81 (0)	79 (0)	0 (0)	6 (0)	1 (0)	0 (0)	50	
Benchmarks based on HyperQB 2.0 (66)	58 (0)	58 (0)	56 (1)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	1 (0)	54 (0)	26 (0)	0 (0)	5 (0)	0 (0)	0 (0)	9	
A-HLTL benchmarks (12)	8 (0)	8 (0)	8 (1)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	8 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	9	
HyperLTL benchmarks (Part 1) (16)	12 (0)	12 (0)	12 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	12 (0)	6 (0)	0 (0)	0 (0)	0 (0)	0 (0)	12	
Default SMT mode (8)	7 (0)	7 (0)	7 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	7 (0)	3 (0)	0 (0)	0 (0)	0 (0)	0 (0)	7	
Witness-producing mode (8)	5 (0)	5 (0)	5 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	5 (0)	3 (0)	0 (0)	0 (0)	0 (0)	0 (0)	5	
HyperLTL benchmarks (Part 2) (34)	34 (0)	34 (0)	34 (0)	0 (0)	0 (0)	0 (0)	0 (0)	1 (0)	5 (0)	34 (0)	20 (0)	0 (0)	5 (0)	0 (0)	0 (0)	34	
Default SMT mode (22)	22 (0)	22 (0)	22 (0)	0 (0)	0 (0)	0 (0)	0 (0)	1 (0)	1 (0)	22 (0)	9 (0)	0 (0)	4 (0)	0 (0)	0 (0)	22	
Witness-producing mode (12)	12 (0)	12 (0)	12 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	4 (0)	12 (0)	11 (0)	0 (0)	1 (0)	0 (0)	0 (0)	12	
Hardware model checking benchmarks (Verilog) (4)	4 (0)	4 (0)	2 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	4	
Benchmarks based on HYENV (53)	53 (0)	53 (0)	17 (0)	0 (0)	19 (0)	0 (0)	21 (0)	22 (0)	0 (0)	27 (0)	53 (0)	0 (0)	1 (0)	1 (0)	0 (0)	53	
HYENV examples (17)	17 (0)	17 (0)	0 (0)	5 (0)	4 (0)	0 (0)	5 (0)	5 (0)	0 (0)	10 (0)	17 (0)	0 (0)	0 (0)	1 (0)	0 (0)	17	
huya benchmarks (fixed observational depth) (5)	5 (0)	5 (0)	0 (0)	5 (0)	4 (0)	0 (0)	5 (0)	5 (0)	0 (0)	6 (0)	5 (0)	0 (0)	0 (0)	0 (0)	0 (0)	5	
ORHLE benchmarks (13)	13 (0)	13 (0)	0 (0)	5 (0)	5 (0)	0 (0)	5 (0)	6 (0)	0 (0)	6 (0)	13 (0)	0 (0)	1 (0)	0 (0)	0 (0)	13	
api-refinement (3)	3 (0)	3 (0)	0 (0)	1 (0)	1 (0)	0 (0)	1 (0)	1 (0)	0 (0)	3 (0)	3 (0)	0 (0)	0 (0)	0 (0)	0 (0)	3	
blackjack (2)	2 (0)	2 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	1 (0)	0 (0)	2 (0)	2 (0)	0 (0)	0 (0)	0 (0)	0 (0)	2	
delimited-release (5)	5 (0)	5 (0)	0 (0)	1 (0)	1 (0)	0 (0)	1 (0)	1 (0)	0 (0)	5 (0)	5 (0)	0 (0)	0 (0)	0 (0)	0 (0)	5	
flaky-tests (1)	1 (0)	1 (0)	0 (0)	1 (0)	1 (0)	0 (0)	1 (0)	1 (0)	0 (0)	1 (0)	1 (0)	0 (0)	1 (0)	0 (0)	0 (0)	1	
gri (2)	2 (0)	2 (0)	0 (0)	2 (0)	2 (0)	0 (0)	2 (0)	2 (0)	0 (0)	2 (0)	2 (0)	0 (0)	0 (0)	0 (0)	0 (0)	2	
HYENV benchmarks (18)	18 (0)	18 (0)	74 (0)	103 (0)	103 (0)	0 (0)	81 (0)	75 (0)	84 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	18	
satisfiability (210)	184 (0)	184 (0)	2 (0)	3 (0)	3 (0)	0 (0)	2 (0)	2 (0)	2 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	192	
ci_od_smtlib (4)	2 (0)	2 (0)	2 (0)	3 (0)	3 (0)	0 (0)	2 (0)	2 (0)	2 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	3	
ie_od_bool_smtlib (6)	3 (0)	3 (0)	3 (0)	3 (0)	3 (0)	0 (0)	3 (0)	3 (0)	3 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	3	
ie_od_smtlib (7)	4 (0)	4 (0)	4 (0)	4 (0)	4 (0)	0 (0)	4 (0)	4 (0)	4 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	4	
od_smtlib (6)	6 (0)	6 (0)	1 (0)	2 (0)	2 (0)	0 (0)	2 (0)	2 (0)	4 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	6	
opacity_smtlib (41)	38 (0)	38 (0)	10 (0)	12 (0)	12 (0)	0 (0)	12 (0)	12 (0)	11 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	38	
qni_smtlib (70)	57 (0)	57 (0)	43 (0)	43 (0)	43 (0)	0 (0)	43 (0)	37 (0)	45 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	64	
qni_t_smtlib (70)	68 (0)	68 (0)	6 (0)	32 (0)	31 (0)	0 (0)	10 (0)	10 (0)	10 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	68	
sym_smtlib (6)	6 (0)	6 (0)	5 (0)	4 (0)	5 (0)	0 (0)	5 (0)	5 (0)	5 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	6	
che-verification (250)	209 (34)	158 (1)	0 (0)	1 (0)	1 (0)	104 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	49 (0)	21 (0)	0 (0)	0 (0)	212	
with predicate abstraction (98)	97 (0)	98 (0)	0 (0)	0 (0)	0 (0)	47 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	13 (0)	0 (0)	0 (0)	98	
full instances (57)	56 (0)	57 (0)	0 (0)	0 (0)	0 (0)	24 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	16 (0)	8 (0)	0 (0)	0 (0)	57	
HYPA's benchmarks beyond k-safety (3)	2 (0)	3 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	3	
HYPA's k-safety benchmarks (3)	3 (0)	3 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	3	
More k-safety benchmarks (1)	1 (0)	1 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	1	
PDSC comparator benchmarks (50)	50 (0)	50 (0)	0 (0)	0 (0)	0 (0)	24 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	14 (0)	7 (0)	0 (0)	0 (0)	50	
simplified instances (41)	41 (0)	41 (0)	0 (0)	0 (0)	0 (0)	23 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	12 (0)	5 (0)	0 (0)	0 (0)	41	
HYPA's k-safety benchmarks (1)	1 (0)	1 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	1	
PDSC comparator benchmarks (40)	40 (0)	40 (0)	0 (0)	0 (0)	0 (0)	23 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	12 (0)	5 (0)	0 (0)	0 (0)	40	
without predicate abstraction (152)	112 (34)	60 (1)	0 (0)	1 (0)	1 (0)	57 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	23 (0)	8 (0)	0 (0)	0 (0)	114	
full instances (101)	63 (23)	34 (1)	0 (0)	0 (0)	0 (0)	25 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	10 (0)	5 (0)	0 (0)	0 (0)	65	
HYENV's escalating benchmarks (33)	6 (6)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	6	
HYPA's benchmarks beyond k-safety (9)	5 (1)	5 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	1 (0)	0 (0)	0 (0)	6	
HYPA's k-safety benchmarks (6)	1 (0)	2 (1)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	2	
More k-safety benchmarks (2)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0	
PDSC comparator benchmarks (51)	51 (16)	27 (0)	0 (0)	0 (0)	0 (0)	25 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	10 (0)	4 (0)	0 (0)	0 (0)	51	
simplified instances (51)	49 (11)	26 (0)	0 (0)	1 (0)	1 (0)	32 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	13 (0)	3 (0)	0 (0)	0 (0)	49	
HYPA's k-safety benchmarks (2)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0	
PDSC comparator benchmarks (49)	49 (11)	26 (0)	0 (0)	1 (0)	1 (0)	32 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	13 (0)	3 (0)	0 (0)	0 (0)	49	

Table 2: Variant of Table 1, but excluding problems from the training set P_{train} of the VAMPIRE portfolios. The first column “ n (c)” gives the name n of the problem group, and the number of problems c in that group. Each cell “ s (u)” in the remaining columns lists that n problem where solved by a given solver in the respective benchmark group, and that u problems where uniquely solved by this solver and no other solver.

Solver	Competition	Version
Z3-ALPHA	SMTCOMP-2025	SMTCOMP Submission z3alpha
CVC5	SMTCOMP-2025	SMTCOMP Submission cvc5
iPROVER	SMTCOMP-2025	SMTCOMP Submission iprover
SMTITP	SMTCOMP-2025	SMTCOMP Submission smtinterpol
SMTRAT	SMTCOMP-2025	SMTCOMP Submission smtrat
YICESQS	SMTCOMP-2025	SMTCOMP Submission yicesQS
ELDARICA	CHCCOMP-2025	2.3 (latest release)
GOLEM	CHCCOMP-2025	0.9.0 (latest release)
LoAT	CHCCOMP-2025	0.9.8 (latest release)
UNIHORN	CHCCOMP-2025	0.3.1 (latest release)
VAMPIRE	CASC-30	5.0.1 (latest release)
Z3-4.12.2	—	4.12.2
YICES-2.6.4	—	2.6.4
VAMPIRE ^g	—	Commit 94b98f5 ; Portfolio with goal annotations [39]
VAMPIRE ⁺	—	Commit 94b98f5 ; Portfolio ignoring goal annotations [39]

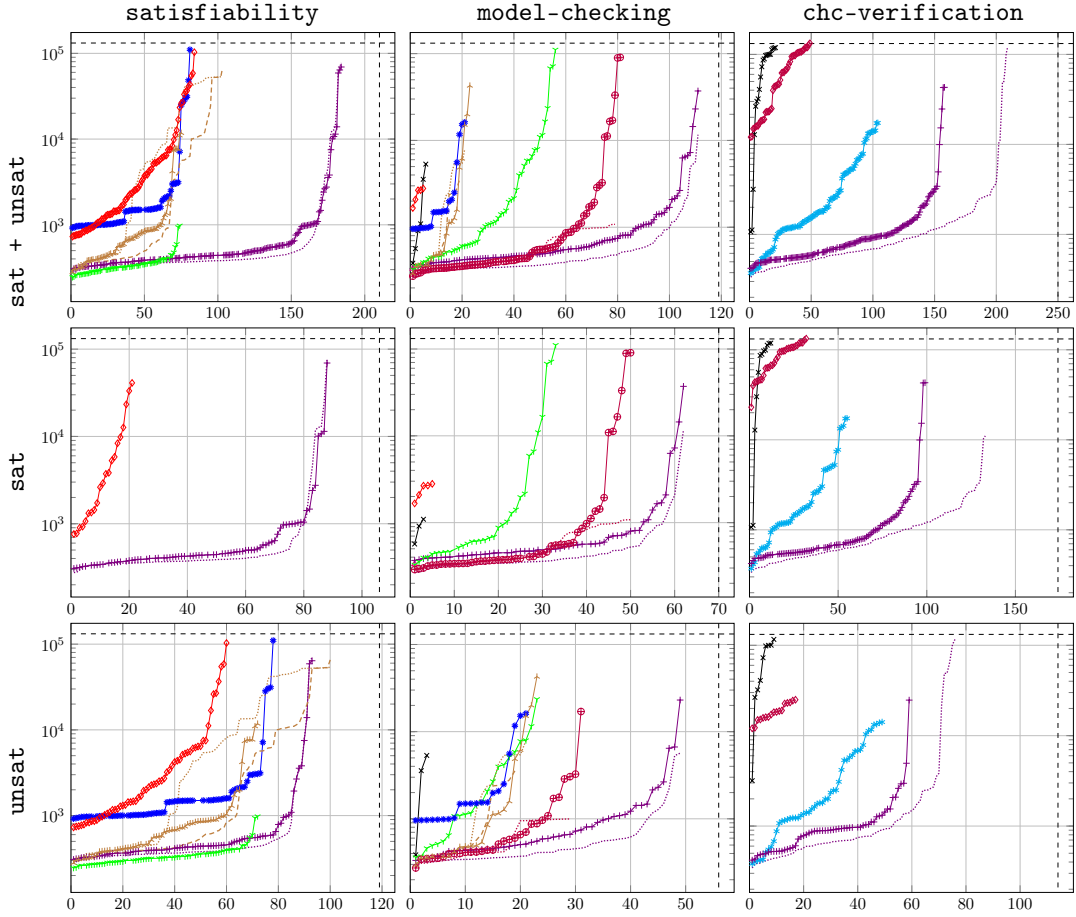


Figure 3: Variant of Fig. 2, but excluding problems from the training set P_{train} of the VAMPIRE portfolios. The y axis shows the time in ms in logarithmic scale. The x axis shows the number of problems solved. A dot in the graph means that a solver solved x problems with a timeout of y ms per problem. In this figure only the test data set P_{test} . The training data P_{train} for vampire portfolio has been filtered out.

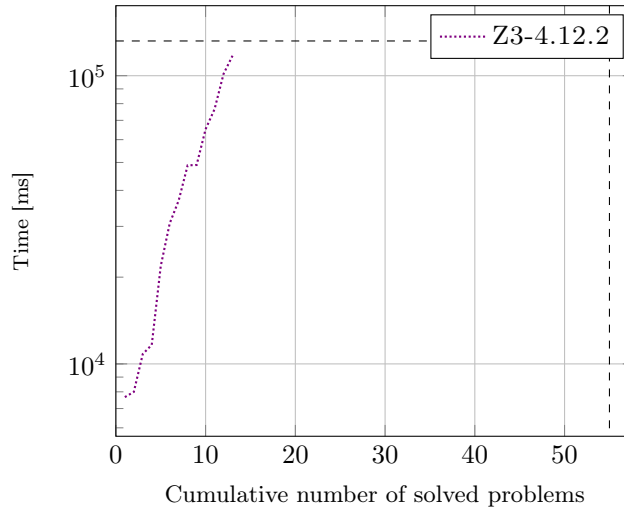


Figure 4: Cumulative number of problems solved in $\text{HY}\exists\forall$'s **escalating** benchmarks. The y axis shows the time in ms in logarithmic scale. The x axis shows the number of problems solved. A dot in the graph means that a solver solved x problems with a timeout of y ms per problem. The horizontal dashed lines show the hard timeout, and the vertical lines show the total number of benchmarks in $\text{HY}\exists\forall$'s **escalating** benchmarks.