Finding counterexamples to ∀∃ hyperproperties

Extended abstract

Tobias Nießen
TU Wien
Institute of Logic and Computation
Vienna, Austria
tobias.niessen@tuwien.ac.at

Georg Weissenbacher
TU Wien
Institute of Logic and Computation
Vienna, Austria
georg.weissenbacher@tuwien.ac.at

Abstract

Verification of software systems against hyperproperties that require quantifier alternation among the quantified trace variables is notoriously difficult because it generally cannot be reduced to the verification of single-trace properties. Such hyperproperties include the class of ∀∃-safety hyperproperties, which contains important hyperproperties such as refinement and generalized non-interference. Existing approaches to the verification of these properties are often incomplete or restricted to finite-state systems. When the hyperproperty does not hold, no existing approaches can fully automatically produce counterexamples demonstrating this fact. We present an algorithm that searches for counterexamples to ∀∃-safety hyperproperties in infinite-state software systems and evaluate its effectiveness based on existing examples from related works.

1 Introduction

Hyperproperties [12] specify properties of systems across multiple execution traces. Many important properties are hyperproperties, for example, observational determinism, injectivity, symmetry, and idempotence are 2-safety properties. In other words, these properties require a statement about two universally quantified trace variables. Some hyperproperties require existential trace quantification and often even quantifier alternation. For example, nondeterminism requires existential quantification of two traces. In this work, we focus on the class of ∀∃-safety hyperproperties. This class contains prominent hyperproperties such as refinement and generalized non-interference.

∀∃ hyperproperties are challenging in multiple ways. Automated verification of such hyperproperties necessarily requires algorithms that are capable of proving statements that contain quantifier alternation. The verification of k-safety hyperproperties can be reduced to single-trace safety verification through self-composition [2] or through product programs [1], and a wide range of approaches exists for the verification of single-trace safety properties. Such a reduction is generally not possible for ∀∃ hyperproperties, so traditional safety verification methods cannot be applied. Instead, a proof of correctness given a ∀∃-safety hyperproperty and a transition system typically requires a witness function for the existentially quantified traces, which is often referred to as a strategy. Synthesizing such a witness function as part of a proof of correctness is 2-EXPTIME-complete for finite-state transition systems [4], and software verification often even requires infinite-state models. In the context of runtime verification, single-trace safety properties and even k-safety hyperproperties can be monitored at runtime to varying degrees. Hyperproperties that existentially quantify over traces, such as ∀∃ hyperproperties, cannot be monitored directly because, for any particular trace that has been obtained at runtime, the witness of the respective existentially quantified trace might never appear at runtime [8].

While formal methods have traditionally focused on the formal verification of an implementation against some specification, the resulting techniques rarely scale to real-world instances. Thus, software engineering commonly relies on underapproximate methods for finding specification violations in implementations. In other words, instead of formally proving the correctness of an implementation with respect to a specification, software engineers use tools to locate – and thus prove the existence of – specification violations. We refer to such methods as underapproximate because a lack of found specification violations generally does not imply that the implementation does not diverge from the specification in some way that has not been discovered by the analysis. Such techniques include testing, fuzzing, symbolic execution, as well as various kinds of static and dynamic analysis.

Such underapproximate methods have recently also been the focus of formal methods and various, often incomplete, methods have been proposed to prove incorrectness instead of correctness. Runtime verification is an inherently underapproximate technique. Recently, Incorrectness Logic [21] has been explored as a natural dual of Hoare Logic for underapproximate reasoning about program behavior. Since then, multiple other logics have been developed that also facilitate underapproximate Hoare-style reasoning. Bounded model checking is underapproximate as well, unless the bound in bounded model checking is sufficiently large to fully explore all possible states of the system [6]. Symbolic execution and concolic testing generally also cannot explore all possible paths in infinite-state systems and are thus underapproximate.

Incorrectness 2024, January 16, 2024, London, United Kingdom
2023.

Verification tools for ∀∃ hyperproperties are often incomplete [5] and might fail even if the property holds. If a ∀∃-safety hyperproperty does not hold, algorithms that attempt to verify such properties generally either do not terminate or, if they do terminate, do not produce counterexamples [4]. In this work, we propose an algorithm for finding counterexamples to ∀∃ hyperproperties. Such algorithms are naturally underapproximate. Because the underlying decision problems are generally undecidable, we do not aim for termination in general. Non-termination of an algorithm that aims to find specification violations does not necessarily correspond to a proof of correctness. We combine symbolic execution for the universally quantified traces with a variant of bounded model checking for the existentially quantified traces.

2 Background

Hyperproperties were introduced by Clarkson and Schneider [12] and relate multiple execution traces of a system.

Bounded model checking is a well-known method for verifying whether a transition system satisfies a trace property. This method is underapproximate insofar that bounded model checking only considers traces of lengths up to some finite bound. Thus, if a trace exists that violates the trace property, its existence will only be detected if the length of the counterexample does not exceed the bound.

Symbolic execution was first introduced during the mid-1970s [7, 19, 20]. Alongside bounded model checking and fuzzing, it has since become one of the most popular techniques for the automated detection of faults in implementation. Several tools based on symbolic execution have been presented to the scientific community, including CUTE [22], DART [17], and KLEE [9]. These tools often combine symbolic and concrete execution in a way that is commonly referred to as concolic testing.

Various logics exist that can represent the semantics of temporal hyperproperties, albeit many are not capable of representing hyperproperties that require quantifier alternation. Observation-based HyperLTL (OHyperLTL) [4] extends HyperLTL [11], which itself is an extension of propositional linear-time temporal logic (LTL). While HyperLTL formulas express properties over multiple traces synchronously, OHyperLTL formulas can represent both synchronous and asynchronous properties by effectively projecting asynchronous traces to synchronous sequences of observations. Additionally, HyperLTL considers traces to be sequences of sets of atomic propositions and thus restricts the atoms within formulas to atomic propositions, whereas atoms in OHyperLTL formulas may contain predicates from the underlying theory. This allows OHyperLTL to effectively reason about variables over infinite domains.

We refer to Beutner and Finkbeiner [4] for the precise definition of the syntax and semantics of OHyperLTL. The original (unbounded) semantics of LTL, HyperLTL, and OHyperLTL are defined for application to infinite traces. To apply the same temporal logics to finite traces, the semantics have to be adapted accordingly. Such bounded semantics for LTL have been defined, for example, by Biere et al. [6], and bounded semantics for HyperLTL have been defined, for example, by Hsu et al. [18]. These bounded semantics for HyperLTL can be adapted for OHyperLTL [4] in a straightforward manner, with some necessary modifications to support quantification over traces that may or may not have a finite number of observation points only.

3 Algorithm

We present an algorithm that, given input programs P and Q and a OHyperLTL [4] specification $\psi = \forall \tau \colon \Omega \,.\, \exists \tau' \colon \xi \,.\, \mathbf{G}(\varphi)$ for some quantifier-free formula φ that is free of temporal operators, searches for a counterexample that witnesses a violation of the given specification.

As is common for bounded model checking, the algorithm repeatedly increments a bound k until a counterexample has been found. Due to the nature of OHyperLTL [4], the bound is not the length of the relevant paths (or traces), but rather the number of observation points along said paths.

For each bound k, the algorithm then iterates over all (feasible) paths in the input program P that contain k observation points as determined by Ω. For each such path, the algorithm constructs the relevant symbolic constraints, which restrict the values of program variables during subsequent SAT/SMT queries. This step is inherent to symbolic execution.

The algorithm then repeatedly queries a SAT or SMT solver with formulas constructed from the specification, from the previously generated symbolic constraints arising from the selected path of P, as well as from an encoding of the relevant behavior of Q. At some point, either all potential counterexamples have been refuted, or some counterexample cannot be refuted. If the latter is true, then a counterexample has been found. Otherwise, eventually, all potential counterexamples along the selected path have been refuted, and the algorithm begins processing the next path. This step resembles bounded model checking of the control flow automaton Q against the specification ψ and additional constraints due to the selected path in P.

The proposed algorithm relies on multiple abstract procedures whose definition depends on the underlying theory. Even if the underlying theory is decidable, some of the decision procedures that are necessary for our variant of bounded model checking might not terminate due to certain characteristics of the behavior of the input programs. We also discover problems in approximating preconditions, similar to those outlined by Cimatti et al. for their APPROX-PREIMAGE procedure [10]. Nevertheless, we precisely define sufficient conditions that result in relative completeness of our approach, i.e., there is a non-trivial class of input programs and specifications for which our algorithm is guaranteed to find a counterexample if one exists. Similarly, we prove soundness of our algorithm in the context of the modified, bounded semantics of OHyperLTL [4]. Lastly, while the core algorithm assumes exactly one universally and one existentially quantified trace variable, we also show how to apply the proposed algorithm to specifications that contain any number of universally and existentially quantified trace variables. This allows our approach to handle ∀∀∃-safety hyperproperties such as generalized non-interference, as well as properties without quantifier alternation, e.g., ∃∃ hyperproperties for non-determinism or flaky tests and ∀∀ hyperproperties for various mathematical properties, including monotonicity, determinism, injectivity, symmetry, anti-symmetry, asymmetry, totality, idempotence, and many more.

4 Experimental evaluation

We implemented the algorithm described above and evaluated it against a diverse set of examples, many of which we drew from related works [4, 14]. These examples include several instances of refinement and generalized non-interference, as well as various other problems.

When the respective ∀∃-safety hyperproperties hold, the algorithm either does not terminate or terminates without producing a counterexample. The latter outcome generally only occurs if all traces of the input program have a finite number of observation points only, for example, because the input program always terminates after a bounded number of steps. For the vast majority of examples, if the ∀∃-safety hyperproperty does not hold, the algorithm terminates within a short amount of time and produces a concrete trace for the universally quantified trace variable that witnesses the violation. A few examples from the literature lead to divergence in the respective control flow automatons. A small number of examples rely on non-linear arithmetic or non-integer data types, which our implementation currently rejects.

5 Related work

Beutner and Finkbeiner developed a game-theoretic approach to the verification of ∀∃-safety properties [4] in infinite-state settings. An earlier game-theoretic approach for ∀∃ hyperproperties by Coenen et al. [13] followed a similar idea but was limited to finite-state systems. Such strategy-based verification methods are generally incomplete, i.e., verification might fail even though the respective property holds, and existing approaches to completeness do not scale beyond very small systems and constrained classes of properties [3, 5]. Additionally, when verification fails, these algorithms generally cannot produce concrete counterexamples that witness the property violation.

More recently, Beutner and Finkbeiner designed an explicit-state model checker for arbitrary HyperLTL properties [5], which is the first *complete* model checker to support both quantifier alternation and arbitrary temporal operators in the LTL body of HyperLTL formulas. However, like earlier automata-based approaches [16], this method is limited to finite-state systems.

Hsu et al. apply bounded model checking to hyperproperties by reducing the verification task to the quantified Boolean formula (QBF) problem [18]. In some sense, our algorithm lifts this approach to arbitrary theories through the use of an SMT solver, yet we do not require the SMT solver's decision procedure to handle quantifier alternation. Additionally, we use symbolic execution to simplify individual SMT queries such that each query corresponds to a single path in the universally quantified program only. Lastly, our approach benefits from the significant advancements in the design and implementation of SMT solvers.

The SMT solver Yices introduced an algorithm for solving ∃∀ queries in version 2.2 to aid in the synthesis of system parameters [15]. Their method is very similar to the inner loop in our algorithm and even provides termination guarantees for some underlying theories (including bitvectors and subsets of integer arithmetic).

Dickerson et al. use a Hoare-style program logic to verify ∀∃ hyperproperties that combines both overapproximate and underapproximate reasoning [14]. This relational logic can reason about programs directly and produces a set of verification conditions for an SMT solver. Upon verification failure, their implementation produces a counterexample. However, as is common for Hoare-style program logics, loops in the input program present a major challenge for this verification approach, whereas symbolic execution and bounded model checking can often identify problems without considering every possible unrolling of a loop. In fact, the implementation of this program logic, ORHLE [14], requires every loop to be annotated with relevant loop invariants.

6 Conclusion

We present an algorithm capable of searching for counterexamples that witness the violation of ∀∃-safety hyperproperties given user-defined input programs for the respective quantified trace variables. We have demonstrated its effectiveness in locating specification violations by evaluating our implementation against various examples from related works, and we have characterized precisely under what circumstances it succeeds (as opposed to diverging).

It is worth noting that the counterexamples produced by the proposed algorithm consist of concrete traces for the universally quantified traces only. Future work includes an extension that also produces an explanation as to why these traces form a counterexample, likely in the form of a proof.

Acknowledgements. This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 101034440.

References

[1] Gilles Barthe, Juan Manuel Crespo, and César Kunz. 2011. Relational Verification Using Product Programs. In *FM 2011: Formal Methods (Lecture Notes in Computer Science)*, Michael Butler and Wolfram Schulte (Eds.). Springer, Berlin, Heidelberg, 200–214. https://doi.org/10.1007/978-3-642-21437-0_17

[2] G. Barthe, P.R. D'Argenio, and T. Rezk. 2004. Secure information flow by self-composition. In *Proceedings. 17th IEEE Computer Security Foundations Workshop, 2004.* IEEE, Pacific Grove, CA, USA, 100–114. https://doi.org/10.1109/CSFW.2004.1310735

[3] Raven Beutner and Bernd Finkbeiner. 2022. Prophecy Variables for Hyperproperty Verification. In *2022 IEEE 35th Computer Security Foundations Symposium (CSF)*. 471–485. https://doi.org/10.1109/CSF54842.2022.9919658

[4] Raven Beutner and Bernd Finkbeiner. 2022. Software Verification of Hyperproperties Beyond k-Safety. In *Computer Aided Verification (Lecture Notes in Computer Science)*, Sharon Shoham and Yakir Vizel (Eds.). Springer International Publishing, Cham, 341–362. https://doi.org/10.1007/978-3-031-13185-1_17

[5] Raven Beutner and Bernd Finkbeiner. 2023. AutoHyper: Explicit-State Model Checking for HyperLTL. In *Tools and Algorithms for the Construction and Analysis of Systems (Lecture Notes in Computer Science)*, Sriram Sankaranarayanan and Natasha Sharygina (Eds.). Springer Nature Switzerland, Cham, 145–163. https://doi.org/10.1007/978-3-031-30823-9_8

[6] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. 1999. Symbolic Model Checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems (Lecture Notes in Computer Science)*, W. Rance Cleaveland (Ed.). Springer, Berlin, Heidelberg, 193–207. https://doi.org/10.1007/3-540-49059-0_14

[7] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. 1975. SELECT—a formal system for testing and debugging programs by symbolic execution. *ACM SIGPLAN Notices* 10, 6 (April 1975), 234–245. https://doi.org/10.1145/390016.808445

[8] Noel Brett, Umair Siddique, and Borzoo Bonakdarpour. 2017. Rewriting-Based Runtime Verification for Alternation-Free HyperLTL. In *Tools and Algorithms for the Construction and Analysis of Systems (Lecture Notes in Computer Science)*, Axel Legay and Tiziana Margaria (Eds.). Springer, Berlin, Heidelberg, 77–93. https://doi.org/10.1007/978-3-662-54580-5_5

[9] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation (OSDI'08)*. USENIX Association, USA, 209–224.

[10] Alessandro Cimatti and Alberto Griggio. 2012. Software Model Checking via IC3. In *Computer Aided Verification (Lecture Notes in Computer Science)*, P. Madhusudan and Sanjit A. Seshia (Eds.). Springer, Berlin, Heidelberg, 277–293. https://doi.org/10.1007/978-3-642-31424-7_23

[11] Michael R. Clarkson, Bernd Finkbeiner, Masoud Koleini, Kristopher K. Micinski, Markus N. Rabe, and César Sánchez. 2014. Temporal Logics for Hyperproperties. In *Principles of Security and Trust (Lecture Notes in Computer Science)*, Martín Abadi and Steve Kremer (Eds.). Springer, Berlin, Heidelberg, 265–284. https://doi.org/10.1007/978-3-642-54792-8_15

[12] Michael R. Clarkson and Fred B. Schneider. 2010. Hyperproperties. *Journal of Computer Security* 18, 6 (Sept. 2010), 1157–1210.

[13] Norine Coenen, Bernd Finkbeiner, César Sánchez, and Leander Tentrup. 2019. Verifying Hyperliveness. In *Computer Aided Verification (Lecture Notes in Computer Science)*, Isil Dillig and Serdar Tasiran (Eds.). Springer International Publishing, Cham, 121–139. https://doi.org/10.1007/978-3-030-25540-4_7

[14] Robert Dickerson, Qianchuan Ye, Michael K. Zhang, and Benjamin Delaware. 2022. RHLE: Modular Deductive Verification of Relational ∀∃ Properties. In *Programming Languages and Systems (Lecture Notes in Computer Science)*, Ilya Sergey (Ed.). Springer Nature Switzerland, Cham, 67–87. https://doi.org/10.1007/978-3-031-21037-2_4

[15] Bruno Dutertre. 2014. Yices 2.2. In *Computer Aided Verification (Lecture Notes in Computer Science)*, Armin Biere and Roderick Bloem (Eds.). Springer International Publishing, Cham, 737–744. https://doi.org/10.1007/978-3-319-08867-9_49

[16] Bernd Finkbeiner, Markus N. Rabe, and César Sánchez. 2015. Algorithms for Model Checking HyperLTL and HyperCTL*. In *Computer Aided Verification (Lecture Notes in Computer Science)*, Daniel Kroening and Corina S. Păsăreanu (Eds.). Springer International Publishing, Cham, 30–48. https://doi.org/10.1007/978-3-319-21690-4_3

[17] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: directed automated random testing. *ACM SIGPLAN Notices* 40, 6 (June 2005), 213–223. https://doi.org/10.1145/1064978.1065036

[18] Tzu-Han Hsu, César Sánchez, and Borzoo Bonakdarpour. 2021. Bounded Model Checking for Hyperproperties. In *Tools and Algorithms for the Construction and Analysis of Systems*, Jan Friso Groote and Kim Guldstrand Larsen (Eds.). Vol. 12651. Springer International Publishing, Cham, 94–112. https://doi.org/10.1007/978-3-030-72016-2_6

[19] James C. King. 1975. A new approach to program testing. *ACM SIGPLAN Notices* 10, 6 (April 1975), 228–233. https://doi.org/10.1145/390016.808444

[20] James C. King. 1976. Symbolic execution and program testing. *Commun. ACM* 19, 7 (July 1976), 385–394. https://doi.org/10.1145/360248.360252

[21] Peter W. O'Hearn. 2020. Incorrectness logic. *Proceedings of the ACM on Programming Languages* 4, POPL (Jan. 2020), 1–32. https://doi.org/10.1145/3371078

[22] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: a concolic unit testing engine for C. *ACM SIGSOFT Software Engineering Notes* 30, 5 (Sept. 2005), 263–272. https://doi.org/10.1145/1095430.1081750