International Journal on Software Tools for Technology Transfer (2020) 22:729–740
https://doi.org/10.1007/s10009-020-00552-5

GENERAL

Special Issue: TACAS 2018

Efficient monitoring of hyperproperties using prefix trees

Bernd Finkbeiner[1] · Christopher Hahn[1] · Marvin Stenger[1] · Leander Tentrup[1]

Published online: 20 February 2020

Abstract
Hyperproperties, such as non-interference and observational determinism, relate multiple computation traces with each other and are thus not monitorable by tools that consider computations in isolation. We present the monitoring approach implemented in the latest version of RVHyper, a runtime verification tool for hyperproperties. The input to the tool are specifications given in the temporal logic HyperLTL, which extends linear-time temporal logic (LTL) with trace quantifiers and trace variables. RVHyper processes execution traces sequentially until a violation of the specification is detected. In this case, a counterexample, in the form of a set of traces, is returned. RVHyper employs a range of optimizations: a preprocessing analysis of the specification and a procedure that minimizes the traces that need to be stored during the monitoring process. In this article, we introduce a novel trace storage technique that arranges the traces in a tree-like structure to exploit partially equal traces. We evaluate RVHyper on existing benchmarks on secure information flow control, error correcting codes, and symmetry in hardware designs. As an example application outside of security, we show how RVHyper can be used to detect spurious dependencies in hardware designs.

Keywords Runtime verification · Monitoring · Hyperproperties

1 Introduction

Hyperproperties [10] are widely studied in (but not limited to) the context of secure information flow control. They generalize trace properties in that they not only check the correctness of *individual* computation traces in isolation, but relate *multiple* computation traces to each other. Examples include information flow policies, such as observational determinism [33,34,42], (quantitative) non-interference [33,36,41] as well as symmetry [26] and spurious dependencies in hardware designs [23], error correcting codes [26], and anti-doping of automotive software [14].

In this article, we present the monitoring approach implemented in the latest version of RVHyper, an automata-based monitoring tool for hyperproperties [24]. In dynamic verification of hyperproperties, efficient and light-weight monitoring techniques are instrumented in systems, which are usually far beyond the scope of static verification approaches. By doing so, countermeasures are enacted before, for example, irreparable information leaks happen. A runtime verification tool for hyperproperties is in particular useful if the implementation of a security critical system is not available. Even without access to the source code, monitoring the observable execution traces still detects insecure information flow. RVHyper also supports the verification workflow by providing a method to test and develop specifications: Specifications can be checked on sample traces without the need for a complete model. Based on the feedback of RVHyper, the specification can be refined until it matches the intended meaning.

This work was partially supported by the German Research Foundation (DFG) as part of the Collaborative Research Center "Methods and Tools for Understanding and Controlling Privacy" (CRC 1223) and the Collaborative Research Center "Foundations of Perspicuous Software Systems" (TRR 248, 389792660), and by the European Research Council (ERC) Grant OSARES (No. 683300).

✉ Christopher Hahn
Hahn@react.uni-saarland.de

Bernd Finkbeiner
Finkbeiner@react.uni-saarland.de

Marvin Stenger
Stenger@react.uni-saarland.de

Leander Tentrup
Tentrup@react.uni-saarland.de

[1] Reactive Systems Group, Saarland University, Saarbrücken, Germany

The input of RVHyper is given in the temporal logic HyperLTL [9], which expresses temporal hyperproperties by extending linear-time temporal logic with *explicit* trace quantification (see [11] for a recent study of hyperlogics). HyperLTL has been used extensively to specify hyperproperties of practical interest (e.g. [14,21,23–26]). For example, observational determinism is expressed as the following formula:

$$\forall\pi.\forall\pi'.(o_\pi = o_{\pi'})\mathcal{W}(i_\pi \neq i_{\pi'}),$$

stating that every trace pair π, π' has to agree on the output as long as it agrees on the inputs as well. When detecting a violation, RVHyper outputs a counterexample, which is a set of traces that does not satisfy the input formula.

Efficient model checking, synthesis, and satisfiability checking tools for HyperLTL already exist [12,19–22,25,26]. Implementing an efficient runtime verification tool for HyperLTL specifications is, despite recent theoretical progress [1,5–7,24,28,29,37], difficult: In principle, the monitor not only needs to process every observed trace, but must also *store* every trace observed so far, so that future traces can be compared with the traces seen so far.

The previous version of RVHyper tackles this challenging problem by implementing two optimizations [23,24]: a *specification analysis* to detect exploitable properties of a hyperproperty, such as *symmetry* and a *trace analysis*, which detects all redundant traces that can be omitted during the monitoring process. A limitation of the trace analysis, which is based on a language inclusion check, is that only *entire* traces can be analyzed and pruned. For example, consider the traces $t_1 = \{a\}\{a\}\{\}$ and $t_2 = \{a\}\{\}\{a\}$ of length 3 and the HyperLTL formula $\forall\pi.\forall\pi'.\square(a_\pi \rightarrow \neg b_{\pi'})$. Neither t_1 nor t_2 is dominated by the other trace, in the sense of the trace analysis, i.e., that one of the traces poses strictly less requirements on future traces [24]. The traces, however, are equal on the first position. This provides an opportunity for optimization, which our new approach exploits. We introduce a novel trace storage technique (that also has massive impact on the running time), such that RVHyper can also handle *partially equal* traces by storing them in a tree structure.

We evaluate RVHyper on existing benchmarks such as classical information flow security by checking for violations of non-interference or monitoring error-resistant encoder. HyperLTL is, however, not limited to security policies. As an example of such an application beyond security, we show how RVHyper can be used to detect spurious dependencies in hardware designs.

Structure of this article The remainder of this article is structured as follows. We begin by giving preliminaries on HyperLTL, its finite trace semantics and notation in Sect. 2. In Sect. 3, we present automata-based monitoring approach implemented in RVHyper, before discussing optimizations in Sect. 4 that make the monitoring feasible in practice. In Sect. 5, we evaluate RVHyper with a focus on the novel storage optimization technique using our tree data structure.

This is a revised and extended version of a paper that appeared at TACAS 2018 [23]. Our contribution and extension compared to [23] is the inclusion of a new trace storage optimization technique presented in Sect. 4.2 and an extended evaluation in Sect. 5.

Related work The temporal logic HyperLTL was introduced to model check security properties of reactive systems [9,26]. For one of its predecessors, SecLTL [16], there has been a proposal for a white-box monitoring approach [17] based on alternating automata. A recent survey on algorithms for monitoring hyperproperties is given in [28]. Agrawal and Bonakdarpour [1] were the first to study the monitoring problem of HyperLTL for the sequential model. They give a syntactic characterization of monitorable HyperLTL formulas. They present a first monitoring algorithm based on a progression logic expressing trace interdependencies and the composition of an LTL_3 monitor. A first constraint-based approach has been outlined in [7], which works for a subclass of HyperLTL specifications. The idea is to identify a set of propositions of interest and store corresponding constraints. A constraint-based algorithm for the complete fragment of $\forall^2$ HyperLTL formulas has been proposed in [29]. The algorithms rewrite a HyperLTL formula and an incoming event into a constraint composed of a plain LTL requirement as well as a HyperLTL requirement. An constraint system is built incrementally: The HyperLTL part is encoded with variables, which will be incrementally defined with more incoming events of a trace. Like with our monitoring algorithm, they do not have access to the implementation (black box), but in contrast to our work, they do not provide witnessing traces as a monitor verdict.

In [5], the authors study the complexity of monitoring hyperproperties. They show that the form and size of the input as well as the formula have a significant impact on the feasibility of the monitoring process. They differentiate between several input forms and study their complexity: a set of linear traces, tree-shaped Kripke structures, and acyclic Kripke structures. For acyclic structures and alternation-free HyperLTL formulas, the problems complexity gets as low as NC.In [6,37], the authors study where static analysis can be combined with runtime verification techniques to monitor HyperLTL formulas beyond the alternation-free fragment.

For certain information flow policies, like non-interference and some extensions, dynamic enforcement mechanisms have been proposed. Techniques for the enforcement of information flow policies include tracking dependencies at the hardware level [38], language-based monitors [2–4,35,40], and abstraction-based dependency tracking [8,27,30]. Secure multi-execution [15] is a technique that can enforce non-interference by executing a program multiple times in differ-

ent security levels. To enforce non-interference, the inputs are replaced by default values whenever a program tries to read from a higher security level.

2 Preliminaries

Let AP be a finite set of *atomic propositions*, and let $\Sigma = 2^{\text{AP}}$ be the corresponding *alphabet*. An infinite *trace* $t \in \Sigma^\omega$ is an infinite sequence over the alphabet. A subset $T \subseteq \Sigma^\omega$ is called a *trace property*. A *hyperproperty* $H \subseteq 2^{(\Sigma^\omega)}$ is a generalization of a trace property. A finite trace $t \in \Sigma^+$ is a finite sequence over Σ. In the case of finite traces, $|t|$ denotes the length of a trace. We use the following notation to access and manipulate traces: Let t be a trace and i be a natural number. $t[i]$ denotes the ith element of t. Therefore, $t[0]$ represents the first element of the trace. Let j be natural number. If $j \geq i$ and $i \leq |t|$, then $t[i, j]$ denotes the sequence $t[i]t[i+1]\cdots t[\min(j, |t|-1)]$. Otherwise, it denotes the empty trace ϵ. $t[i\rangle$ denotes the suffix of t starting at position i. For two finite traces s and t, we denote their concatenation by $s \cdot t$.

HyperLTL syntax HyperLTL [9] extends LTL with trace variables and trace quantifiers. Let $\mathcal{V}$ be a finite set of trace variables. The syntax of HyperLTL is given by the grammar

$$\varphi := \forall\pi.\ \varphi \mid \exists\pi.\ \varphi \mid \psi$$
$$\psi := a_\pi \mid \psi \wedge \psi \mid \neg\psi \mid \bigcirc\psi \mid \psi\ \mathcal{U}\ \psi,$$

where $a \in \text{AP}$ is an atomic proposition and $\pi \in \mathcal{V}$ is a trace variable. Atomic propositions are indexed by trace variables. The explicit trace quantification enables us to express properties like "on all traces φ must hold," expressed by $\forall\pi.\varphi$. Dually, we can express "there exists a trace such that φ holds," expressed by $\exists\pi.\varphi$. We use the standard derived operators *release* $\varphi\mathcal{R}\psi := \neg(\neg\varphi\mathcal{U}\neg\psi)$, *eventually* $\Diamond\varphi := \text{true}\mathcal{U}\varphi$, *globally* $\Box\varphi := \neg\Diamond\neg\varphi$, and *weak until* $\varphi_1\mathcal{W}\varphi_2 := (\varphi_1\mathcal{U}\varphi_2)\vee\Box\varphi_1$. As we use the finite trace semantics, $\bigcirc\varphi$ denotes the *strong* version of the next operator, i.e., if a trace ends before the satisfaction of φ can be determined, the satisfaction relation, defined below, evaluates to false. To enable duality in the finite trace setting, we additionally use the *weak* next operator $\tilde{\bigcirc}\varphi$ which evaluates to true if a trace ends before the satisfaction of φ can be determined and is defined as $\tilde{\bigcirc}\varphi := \neg\bigcirc\neg\varphi$. We call ψ of a HyperLTL formula $\mathbf{Q}.\psi$, with an arbitrary quantifier prefix $\mathbf{Q}$, the *body* of the formula. A HyperLTL formula $\mathbf{Q}.\psi$ is in the *alternation-free fragment* if either $\mathbf{Q}$ consists solely of universal quantifiers or solely of existential quantifiers. We also denote the respective alternation-free fragments as the $\forall^n$ fragment and the $\exists^n$ fragment, with n being the number of quantifiers in the prefix.

Finite trace semantics We recap the finite trace semantics for HyperLTL [7], which is itself based on the finite trace semantics of LTL [32]. Let $\Pi_{\text{fin}} : \mathcal{V} \to \Sigma^+$ be a partial function mapping trace variables to finite traces. We define $\epsilon[0]$ as the empty set. $\Pi_{\text{fin}}[i\rangle$ denotes the trace assignment that is equal to $\Pi_{\text{fin}}(\pi)[i\rangle$ for all $\pi \in \text{dom}(\Pi_{\text{fin}})$. By slight abuse of notation, we write $t \in \Pi_{\text{fin}}$ to access traces t in the image of Π_{fin}. The satisfaction of a HyperLTL formula φ over a finite trace assignment Π_{fin} and a set of finite traces T, denoted by $\Pi_{\text{fin}} \vDash_T \varphi$, is defined as follows:

$\Pi_{\text{fin}} \vDash_T a_\pi$	if $a \in \Pi_{\text{fin}}(\pi)[0]$
$\Pi_{\text{fin}} \vDash_T \neg\varphi$	if $\Pi_{\text{fin}} \nvDash_T \varphi$
$\Pi_{\text{fin}} \vDash_T \varphi \vee \psi$	if $\Pi_{\text{fin}} \vDash_T \varphi$ or $\Pi_{\text{fin}} \vDash_T \psi$
$\Pi_{\text{fin}} \vDash_T \bigcirc\varphi$	if $\forall t \in \Pi_{\text{fin}}.\ \|t\| > 1$ and $\Pi_{\text{fin}}[1\rangle \vDash_T \varphi$
$\Pi_{\text{fin}} \vDash_T \varphi\mathcal{U}\psi$	if $\exists i < \min_{t\in\Pi_{\text{fin}}} \|t\|.\ \Pi_{\text{fin}}[i\rangle \vDash_T \psi$
	$\wedge\forall j < i.\ \Pi_{\text{fin}}[j\rangle \vDash_T \varphi$
$\Pi_{\text{fin}} \vDash_T \exists\pi.\varphi$	if $\exists t \in T$ such that $\Pi_{\text{fin}}[\pi \mapsto t] \vDash_T \varphi$
$\Pi_{\text{fin}} \vDash_T \forall\pi.\varphi$	if $\forall t \in T$ holds that $\Pi_{\text{fin}}[\pi \mapsto t] \vDash_T \varphi$

The hyperproperty represented by a HyperLTL formula φ, denoted by $\mathcal{H}(\varphi)$, is the set $\{T \subseteq \Sigma^\omega \mid T \vDash \varphi\}$.

3 Runtime verification of hyperproperties with RVHyper

In this section, we present an overview over RVHyper, before describing the implementation setup, present the monitoring algorithm, and discuss our optimization techniques.

The input of RVHyper is given as a universally quantified HyperLTL formula and, in addition, the observed behavior of the system under consideration. The observed behavior is represented as a trace set T, where each $t \in T$ represents a previously observed execution of the system to monitor. RVHyper can therefore detect violations of every monitorable k-safety hyperproperty (see [24] for an extensive study of monitorability of hyperproperties). If RVHyper detects that the system violates the hyperproperty, it outputs a counterexample, i.e., a k-ary tuple of traces, where k is the number of quantifiers in the HyperLTL formula.

3.1 Implementation details

RVHyper[1] is written in C++. We use *spot* [18] for building the deterministic monitor automata and the *Buddy* BDD library for handling symbolic constraints. We use the HyperLTL satisfiability solver EAHyper [19,22] to determine whether the input formula is reflexive, symmetric, or transitive.

[1] The implementation is available at https://react.uni-saarland.de/tools/rvhyper/.

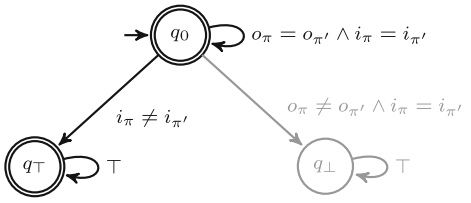


Fig. 1 Visualization of a monitor template corresponding to the formula given in Example 1. We use a symbolic representation of the transition function δ. We depict a sink rejecting state $q_\bot$

Depending on those results, we omit redundant tuples in the monitoring algorithm.

3.2 Online monitoring algorithm

For the online algorithm, we use standard techniques for building LTL monitoring automata and use this to instantiate this monitor by the traces as specified by the HyperLTL formula. Let AP be a set of atomic propositions and $\mathcal{V} = \{\pi_1, \ldots, \pi_n\}$ a set of trace variables. A deterministic monitor template $\mathcal{M} = (\Sigma, Q, \delta, q_0, F)$ is a tuple of a finite alphabet $\Sigma = 2^{(\mathrm{AP}\times\mathcal{V})}$, a non-empty set of states Q, a partial transition function $\delta : Q \times \Sigma \hookrightarrow Q$, a designated initial state $q_0 \in Q$, and a set of accepting states $F \subseteq Q$. The instantiated automaton runs in parallel over traces in $(2^{\mathrm{AP}})^*$, and thus, we define a run with respect to a n-ary tuple $N \in ((2^{\mathrm{AP}})^*)^n$ of finite traces. A run of N is a sequence of states $q_0 q_1 \cdots q_m \in Q^*$, where m is the length of the smallest trace in N, starting in the initial state q_0 such that for all i with $0 \leq i < m$, it holds that

$$\delta\left(q_i, \bigcup_{j=1}^{n} \bigcup_{a \in N(j)(i)} \{(a, \pi_j)\}\right) = q_{i+1}.$$

A tuple N is accepted if there is a run on $\mathcal{M}$ that ends in an accepting state. For LTL, such a deterministic monitor can be constructed in doubly exponential time in the size of the formula [13,39].

Example 1 As an example formula, consider again the observational determinism formula introduced in the introduction:

$$\forall\pi.\forall\pi'.(o_\pi = o_{\pi'}) \mathcal{W} (i_\pi \neq i_{\pi'}),$$

The corresponding monitor template is depicted in Fig. 1.

The algorithm for monitoring HyperLTL formulas when traces are given sequentially to the monitor is presented as Algorithm 1. After building the deterministic monitoring automaton $\mathcal{M}_\varphi$, the algorithm accepts new traces and afterward proceeds with the pace of the incoming stream. We have a variable S that maps tuples of traces to states of the deterministic monitor. Whenever the current trace t progresses, we progress every tuple $(t_1, \ldots, t_n)$ that contains t with one of the following outcomes:

1. One of the traces $t_1, \ldots, t_n$ may have ended, and thus, we check if the monitor is in an accepting state and report a violation if this is not the case.
2. There is a successor state in the monitor, and thus, we update S.
3. There is no successor state, and hence, we report a violation.

When a new trace t starts, only new tuples are considered for S that are tuples $\mathbf{t} \in (T \cup \{t\})^n$ containing the new trace t.

input : $\forall^n$ HyperLTL formula φ
output: satisfied or n-ary tuple witnessing violation

$\mathcal{M}_\varphi = (\Sigma_\mathcal{V}, Q, q_0, \delta, F) =$ `build_template(`φ`)`;
$T \leftarrow \emptyset$;
$S : T^n \to Q$ // initially empty;
t // container for the subsequently incoming event traces
while *there is a new event trace* **do**
  $t \leftarrow \epsilon$ // initialize empty trace;
  for $\mathbf{t} \in ((T \cup \{t\})^n \setminus T^n)$ **do** init S for every new tuple $\mathbf{t}$
    $S(\mathbf{t}) \leftarrow q_0$;
  end
  while $p \in \Sigma$ *is a new input event* **do**
    $t' \leftarrow t\ p$ // append p to t;
    for $((t_1, \ldots, t_n), q) \in S$ *where* $t \in \{t_1, \ldots, t_n\}$ **do** // progress every state in S
      if $\exists t^* \in \{t_1, \ldots, t_n\}.\ |t^*| < |t|$ **then** // some trace ended
        if $q \in F$ **then**
          remove $(t_1, \ldots, t_n)$ from S;
          continue;
        else
          return violation and witnessing tuple $(t_1, \ldots, t_n)$;
        end
      $\mathbf{t} \leftarrow (t_1, \ldots, t_n)$, where t_i is replaced with t', for each $t_i = t$;
      if $\delta(q, \bigcup_{i=1}^{n} \bigcup_{a \in \mathbf{t}(i)[|t'|-1]} \{(a, \pi_i)\}) = q'$ **then**
        $S(\mathbf{t}) \leftarrow q'$;
      else
        return violation and witnessing tuple $\mathbf{t}$;
      end
    end
    $t \leftarrow t'$ // re-assign t;
  end
  $T \leftarrow T \cup \{t\}$;
  //add t to already seen traces
end
return satisfied;

Algorithm 1: Online monitoring algorithm for $\forall^n$HyperLTL: the algorithm subsequently reads event traces and monitors them against already seen traces. Already seen traces are stored in T before continuing with a new event trace.

Example 2 We continue Example 1 by showing how the algorithm progresses on the given formula. Assume for the sake of readability that we have a single input proposition i and a single output proposition o. Furthermore, assume that we have already seen the trace $t_0 = \{i\}\{i, o\}\{o\}$, that is, $T = \{t_0\}$ and $S(t_0, t_0) = q_0$. We now show how the algorithm continues with a fresh trace t_1. In lines 6–8, we add the pairs (t_0, t_1), (t_0, t_1), and (t_1, t_1) with the initial state q_0 to S. Let $p = \{i\}$ be the first input proposition; thus, $t_1 = \{i\}$. Since $t_0[0] = t_1[0]$, the monitor remains in q_0 for every tuple. Let $p = \{i\}$ be the next input proposition; thus, $t_1 = \{i\}\{i\}$. Consider the tuple (t_0, t_1). As $t_0[1]$ and $t_1[1]$ are equal with respect to i but differ in o, the monitor progresses to the rejecting state $q_\bot$ and the algorithm terminates by reporting the violation. If the previous input proposition is $p = \{\}$, both tuples (t_0, t_1) and (t_1, t_0) would progress to the accepting sink state $q_\top$ as the input proposition is different to $t_0[1]$. Assume two more inputs, e.g., $t_1 = \{i\}\{i\}\{\}\{\}$; then the pairs (t_0, t_0), (t_0, t_1), and (t_1, t_0) are removed from S as t_1 is strict longer than t_0.

4 Optimizations

In this section, we present three optimizations implemented in RVHyper, which, as we will see in the evaluation section, are necessary to make the automata-based monitoring approach feasible in practice. We begin by explaining a *specification analysis*, which is a preprocessing step that exploits properties of the specification to reduce the algorithmic workload. In the subsequent section, we show how RVHyper tackles the problem of potentially unbounded memory consumption: We recap the *trace analysis*, which was so far the only storage optimization implemented in RVHyper. We then provide a novel storage optimization technique based on prefix trees, so-called *tries*, to exploit partial equality of the given traces.

4.1 Specification analysis

In the example execution in Example 2, we have seen that the algorithm had to do more work than necessary to monitor observational determinism. For example, a tuple (t, t) for some trace t cannot violate observational determinism as the traces are equal. Further, from the pairs (t, t') and (t', t) for some traces t and t', we only need to check one of them as a violation in one of them implies the violation in the other pair. We can automatically check for such conditions and, thus, omit unnecessary work.

RVHyper implements the specification analysis with the HyperLTL satisfiability solver EAHyper [19,22]. The *specification analysis* is a preprocessing step that analyzes the HyperLTL formula under consideration. EAHyper can detect whether a formula is (1) *symmetric*, i.e., we halve the number of instantiated monitors, (2) *transitive*, i.e., we reduce the number of instantiated monitors to two, or (3) *reflexive*, i.e., we can omit the self-comparison of traces.

Definition 1 [24] Let ψ be the quantifier-free part of some HyperLTL formula φ over trace variables $\mathcal{V}$. We say φ is invariant under trace variable permutation $\sigma : \mathcal{V} \to \mathcal{V}$, if for any set of traces $T \subseteq \Sigma^\omega$ and any assignment $\Pi : \mathcal{V} \to T$, $(\emptyset, \Pi, 0) \vDash \psi \Leftrightarrow (\emptyset, \Pi \circ \sigma, 0) \vDash \psi$. We say φ is symmetric, if it is invariant under every trace variable permutation in $\mathcal{V}$.

Symmetry is particularly interesting since many information flow policies satisfy this property. Consider, for example, observational determinism: $\forall\pi.\forall\pi'.(o_\pi = o_{\pi'})\mathcal{W}(i_\pi \neq i_{\pi'})$. RVHyper detects symmetry by translating this formula to a formula that is unsatisfiable if there exists no set of traces for which every trace pair violates the symmetry condition: $\exists\pi.\exists\pi'.\big((o_\pi = o_{\pi'})\mathcal{W}(i_\pi \neq i_{\pi'})\big) \nleftrightarrow \big((o_\pi = o_{\pi'})\mathcal{W}(i_\pi \neq i_{\pi'})\big)$. If the resulting formula turns out to be unsatisfiable, RVHyper omits the symmetric instantiations of the monitor automaton.

Definition 2 [24] Let ψ be the quantifier-free part of some HyperLTL formula φ over trace variables $\{\pi_1, \pi_2\}$. Let $T = \{t_1, t_2, t_3\} \in \Sigma^\omega$ be a three-elemented set of traces. We define the assignment $\Pi_{i,j} : \mathcal{V} \to \Sigma^\omega$ by $\Pi_{i,j} := \{\pi_1 \mapsto t_i, \pi_2 \mapsto t_j\}$. We say φ is transitive, if for all three-elemented sets T it holds that $(\emptyset, \Pi_{1,2}, 0) \vDash \psi \wedge (\emptyset, \Pi_{2,3}, 0) \vDash \psi \to (\emptyset, \Pi_{1,3}, 0) \vDash \psi$.

While symmetric HyperLTL formulas allow us to prune half of the monitor instances, transitivity of a HyperLTL formula has an even larger impact on the required memory. Equality, i.e., $\forall\pi.\forall\pi'.\square(a_\pi \leftrightarrow a_{\pi'})$, for example, is transitive and symmetric and allows us to reduce the number of monitor instances to one, since we can check equality against any reference trace.

Definition 3 [24] Let ψ be the quantifier-free part of some HyperLTL formula φ over trace variables $\mathcal{V}$. We say φ is reflexive, if for any trace $t \in \Sigma^\omega$ and the corresponding assignment $\Pi : \mathcal{V} \to \{t\}$, $(\emptyset, \Pi, 0) \vDash \psi$.

Lastly, if a formula is reflexive, RVHyper omits the composition of a trace with itself during the monitoring process. For example, equality and observational determinism have reflexive HyperLTL formulas.

Example 3 Consider again the observational determinism formula from Example 1. We have seen that this formula is both, reflexive and symmetric; thus, we can omit those instances in the algorithm.

4.2 Optimizing trace storage

The main obstacle in monitoring hyperproperties is the potentially unbounded space consumption. Previously, RVHyper

employed a *trace analysis* technique to detect redundant traces, with respect to a given HyperLTL formula, i.e., traces that can be safely discarded without losing any information and without losing the ability to return a counterexample.

Definition 4 [24] Given a HyperLTL formula φ, a trace set T, and an arbitrary $t \in TR$, we say that t is (T, φ)-redundant if T is a model of φ if and only if $T \cup \{t\}$ is a model of φ as well, formally

$$\forall T' \supseteq T.\, T' \in \mathcal{H}(\varphi) \Leftrightarrow T' \cup \{t\} \in \mathcal{H}(\varphi).$$

Definition 5 [24] Given $t, t' \in \Sigma^\omega$, we say t *dominates* t' with respect to φ (or simply t *dominates* t' if it is clear from the context) if t' is $(\{t\}, \varphi)$-redundant.

Example 4 For observational determinism, a trace t is dominated by a trace t' if $|t| < |t'|$ and both traces agree on the input propositions.

This is efficiently implemented in RVHyper (cf. Algorithm 2) and is guaranteed to catch all redundant traces. In our experiments [23,24], we made the observation that traces often share the same prefixes, leading to a lot of redundant monitor automaton instantiations, repetitive computations, and duplicated information when those traces get stored.

The trace analysis, as it is based on a language inclusion check of the entire traces, cannot handle *partial redundancy*, for example, in the case that traces have redundant prefix requirements. This leaves room for optimization, which we address by implementing a *trie* data structure for managing the storage of incoming traces.

input : HyperLTL formula φ, redundancy free trace set T, fresh trace t
output: redundancy free set of traces $T_{min} \subseteq T \cup \{t\}$

```
M_φ = build_template(φ)
foreach t' ∈ T do
    if t' dominates t then
        return T;
    end
end
foreach t' ∈ T do
    if t dominates t' then
        T := T \ {t'};
    end
end
return T ∪ {t};
```

Algorithm 2: Trace analysis algorithm to minimize trace storage.

Tries, also known as prefix trees, are a tree data structure, which can represent a set of words over an alphabet in a compact manner. The root of a trie is identified with the empty word ϵ; additionally, each node can have several child nodes, each of which corresponds to a unique letter getting appended to the representing word of the parent node. So the set of words of a trie is identified with the set of words the leaf nodes represent.

Definition 6 A trie is a four tuple $(\Sigma, \mathcal{T}, \longrightarrow, \tau_0)$ consisting of

- A finite alphabet Σ,
- A non-empty set of states $\mathcal{T}$,
- A transition function $\longrightarrow: \mathcal{T} \times \Sigma \to \mathcal{T}$,
- And a designated initial state $\tau_0 \in \mathcal{T}$ called the root.

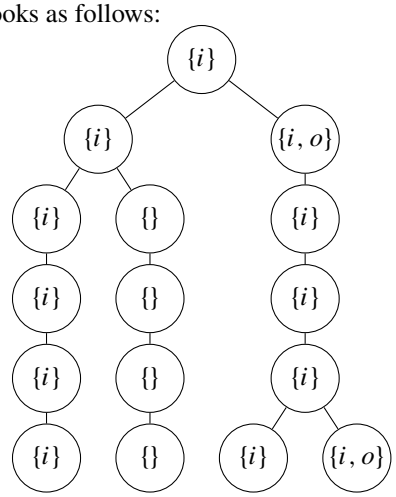
Instead of $((\tau, a), \tau') \in \longrightarrow$, we will write $\tau \xrightarrow{a} \tau'$ in the following. For a trie to be of valid form, we restrict $\longrightarrow$ such that, $\forall \tau, \tau' \in \mathcal{T}.|\{\tau \xrightarrow{a} \tau' | a \in \Sigma\}| \leq 1$.

In our case, the alphabet would be the set of propositions used in the specification, and the word built by the trie represents the traces. Instead of storing each trace individually, we store all of them in one trie structure, branching only in case of deviation. This means equal prefixes only have to be stored once. Besides the obvious benefits for memory, we also can make use of the maintained trie data structure to improve the runtime of our monitoring algorithms. As traces with same prefixes end up corresponding to the same path in the trie, we only have to instantiate the monitor automaton as much as the trie contains branches.

Example 5 Consider the following traces of length 6 over the alphabet $2^{\{i,o\}}$:

- $t_1 : \{\{i\}, \{i, o\}, \{i\}, \{i\}, \{i\}, \{i, o\}\}$
- $t_2 : \{\{i\}, \{i, o\}, \{i\}, \{i\}, \{i\}, \{i\}\}$
- $t_3 : \{\{i\}, \{i\}, \{i\}, \{i\}, \{i\}, \{i\}\}$
- $t_4 : \{\{i\}, \{i\}, \{\}, \{\}, \{\}, \{\}\}$

After processing the traces sequentially, the resulting trie looks as follows:



4.3 Trie-based monitoring algorithm

We depict a trie-based offline monitoring algorithm in Fig. 3. For the sake of readability, we assume that there are as many traces as universal quantifiers that we progress through all traces in parallel, and that all traces have the same length. This is merely a simplification in the presentation, and one can build the trie in a sequential fashion for online monitoring by a slight modification of the presented algorithm.

Without using tries, our monitoring algorithm was based on instantiating the deterministic monitor template $\mathcal{M}_\varphi$ with tuples of traces. Now, we instantiate $\mathcal{M}_\varphi$ with tuples of tries. Initially, we only have to create the single instance having the root of our trie.

input : $\forall^n$ HyperLTL formula φ
output: satisfied or n-ary tuple witnessing violation

1 $\mathcal{M}_\varphi = (\Sigma_\mathcal{V}, Q, q_0, \delta, F) =$ `build_template(`φ`)`
2 $S : \mathcal{T}^n \to Q$
3 τ_0 :=`new_trie()`
4 $\mathbf{i} := (\tau_0, \ldots, \tau_0) \in \mathcal{T}^n$
5 $I := \{\mathbf{i}\}$ // set of not-yet terminated branches
6 **while** $\boldsymbol{p} \leftarrow$ *new event (in* Σ^n*)* **do**
7 **for** $1 \le j \le n$ **do**
8 $\mathbf{i}(j)$ ←`add_child(`$\mathbf{i}(j)$`,` $\mathbf{p}(j)$`)` // add child with value $\mathbf{p}(j)$ to $\mathbf{i}(j)$ if needed
9 **end**
10 // update set of active branches
11 $I \leftarrow \bigcup_{\mathbf{i} \in I} \{(i'_1, \ldots, i'_n) \mid \mathbf{i}(j) \xrightarrow{a} i'_j, a \in \Sigma, 1 \le j \le n\}$
12 **foreach** $i \in I$ **do**
13 progress every state in S according to δ;
14 **if** *violation in* $\mathcal{M}_\varphi$ **then**
15 // return sequence from root to $\mathbf{i}$
16 $t \leftarrow$ (`rooted_sequence(`$\mathbf{i}(1)$`),...,rooted_sequence(`$\mathbf{i}(n)$`)`)
17 **return** witnessing tuple t^n
18 **end**
19 **end**
20 **end**
21 **return** satisfied if $\forall \mathbf{i} \in I \ldots S(\mathbf{i}) \in F$ else violation

Algorithm 3: Offline algorithm using trie data structure.

The trie-based algorithm has much in common with its previously discussed trace-based counterpart. Initially, we have to build the deterministic monitor automaton $\mathcal{M}_\varphi = (\Sigma_\mathcal{V}, Q, q_0, \delta, F)$. We instantiate the monitor with a fresh trie root τ_0. A mapping from trie instantiations to a state in $\mathcal{M}_\varphi$ $S : \mathcal{T}^n \to Q$ stores the current state of every active branch of the trie, stored in the set I. For each of the incoming traces, we provide an entry in a tuple of tries τ, and each entry gets initialized to τ_0. During the run of our algorithm, these entries are updated such that they always correspond to the word built by the traces up to this point. For as long as there are traces left, which have not yet ended, and we have not yet detected a violation, we will proceed updating the entries in $\mathbf{i}$ as follows. Having entry τ and the correspond trace sequence proceeds with a, if $\exists \tau' \in \mathcal{T}.\tau \xrightarrow{a} \tau'$, we update the entry to τ'; otherwise, we create such a child node of τ (`add_child` in line 8). Creating a new node in the trie always occurs when the prefix of the incoming trace starts to differ from already seen prefixes. After having moved one step in our traces sequences, we have to reflect this step in our trie structure, in order for the trie-instantiated automata to correctly monitor the new propositions. As a trie node can branch to multiple child nodes, each monitor instantiation is replaced by the set of instantiations, where all possible child combinations of the different assigned tries are existent (update of I in line 11). Afterward, we update S in the same way as in Algorithm 2; thus, we omit algorithmic details here. If a violation is detected here, that is, there is no transition in the monitor corresponding to $\mathbf{i}$, we will return the corresponding counterexample as a tuple of traces, as those can get reconstructed by stepping upwards in the tries of $\mathbf{i}$. If the traces end, we check if every open branch $\mathbf{i} \in I$ is in an accepting state.

5 Evaluation

In the following, we evaluate the new version of RVHyper, especially the novel trace storage optimization. We use several benchmarks: an encoder that guarantees a Hamming distance of 2, violations of non-interference on randomly generated traces, and a symmetry property on an implementation of the Bakery protocol. As an example how RVHyper can be used outside security runtime verification, we give a case study on detecting spurious dependencies in hardware designs.

5.1 Error correcting codes

We monitored whether an encoder preserves a Hamming distance of 2. We randomly built traces of length 50. In each position of the trace, the corresponding bit had a 1% chance to be flipped. The specification can be encoded as the following HyperLTL formula [26]:

$$\forall \pi \pi'.(\Diamond(i_\pi \nleftrightarrow i_{\pi'}) \to ((o_\pi \leftrightarrow o_{\pi'}) \mathcal{U}((o_\pi \nleftrightarrow o_{\pi'}) \wedge \bigcirc((o_\pi \leftrightarrow o_{\pi'}) \mathcal{U}(o_\pi \nleftrightarrow o_{\pi'})))))).$$

The right plot of Fig. 2 shows the results of our experiments. We compared the naive monitoring approach to different combinations of RVHyper's optimizations. The specification analysis returns in under one second with the result that the formula is symmetric and reflexive. Hence, as expected, this preprocessing step has a major impact on the running time of the monitoring process as more than half of the, in general

necessary, monitor instantiations can be omitted. A combination of the specification and trace analysis performs nearly equally well as naively storing the traces in our trie data structure. Combining the trie data structure with the specification analysis performs best and results in a tremendous speedup compared to the naive approach.

5.2 Checking non-interference

Non-interference [33] is an important information flow policy demanding that an observer of a system cannot infer any high security input of a system by observing only low security input and output. Formally, we specify that all low security outputs $\mathbf{o}^{\text{low}}$ have to be equal on all system executions as long as the low security inputs $\mathbf{i}^{\text{low}}$ of those executions are the same: $\forall \pi, \pi'. (\mathbf{o}_{\pi}^{\text{low}} \leftrightarrow \mathbf{o}_{\pi'}^{\text{low}}) \mathcal{W} (\mathbf{i}_{\pi}^{\text{low}} \not\leftrightarrow \mathbf{i}_{\pi'}^{\text{low}})$. This class of benchmarks has previously been used to evaluate RVHyper [23]. We repeated the experiments, to show that using the trie data structure is a valid optimization. The results are depicted in Table 1. We chose a trace length of 50 and monitored non-interference on 2000 randomly generated traces, where we distinguish between an input range of 8 to 64 bits. The results show that the trie optimization has an enormous impact compared to a naive approach that solely relies on the specification analysis. As expected, the difference in runtime is especially high on experiments where traces collapse heavily in the trie data structure, i.e., producing almost no instances that must be considered during the monitoring process.

5.3 Symmetry in mutual exclusion protocols

In this benchmark (introduced as a case study in [26]), we monitor whether a Verilog implementation of the Bakery protocol [31] from the VIS verification benchmark satisfies a symmetry property. Symmetry violations indicate that certain clients are privileged. The Bakery protocol is a classical protocol implementing mutual exclusion, working as follows: Every process that wishes to access a critical resource draws a ticket, which is consecutively numbered. The process with the smallest number may access the resource first. If two processes draw a ticket concurrently, i.e., obtaining the same number, the process with the smaller process ID may access the resource first. We monitored the following HyperLTL formula [26]:

$$\forall \pi. \forall \pi'. \square(\text{sym}(\text{select}_{\pi}, \text{select}_{\pi'}) \wedge \text{pause}_{\pi} = \text{pause}_{\pi'}) \\ \rightarrow \square(\text{pc}(0)_{\pi} = \text{pc}(1)_{\pi'} \wedge \text{pc}(1)_{\pi} = \text{pc}(0)_{\pi'}),$$

where select indicates the process ID that runs in the next step and pause indicates whether the step is stuttering. Each process i has a program counter $\text{pc}(i)$ and when process i is selected, $\text{pc}(i)$ is executed. $\text{sym}(\text{select}_{\pi}, \text{select}_{\pi'})$ states that process 0 is selected on trace π and process 1 is selected on trace π'. Unsurprisingly, the implementation violates the specification, as it is provably impossible to implement a mutual exclusion protocol that is entirely symmetric [32]. Figure 3 shows the results of our experiment. In this benchmark, we can observe that the language inclusion check, on which the trace optimization is based on, produces an overhead during the monitoring. Since the traces differ a lot, the trace analysis cannot prune enough traces to be valuable. As there are only a few instances (in this case 4), the trie optimization outperforms the previous version of RVHyper massively on such a low instance count. The specification analysis, however, is always a valuable optimization.

5.4 Case study: detecting spurious dependencies in hardware designs

While HyperLTL has been applied to a range of domains, including security and information flow properties, we focus in the following on a classical verification problem, the independence of signals in hardware designs. We demonstrate how RVHyper can automatically detect such dependencies from traces generated from hardware designs.

Input and output The input to RVHyper is a set of traces and a HyperLTL formula. For the following experiments, we generate a set of traces from the Verilog description of several example circuits by random simulation. If a set of traces violates the specification, RVHyper returns a counterexample.

Specification We consider the problem of detecting whether input signals influence output signals in hardware designs. We write $\mathbf{i} \not\rightsquigarrow \mathbf{o}$ to denote that the inputs $\mathbf{i}$ do not influence the outputs $\mathbf{o}$. Formally, we specify this property as the following HyperLTL formula:

$$\forall \pi_1 \forall \pi_2. (\mathbf{o}_{\pi_1} = \mathbf{o}_{\pi_2}) \mathcal{W} (\bar{\mathbf{i}}_{\pi_1} \neq \bar{\mathbf{i}}_{\pi_2}),$$

where $\bar{\mathbf{i}}$ denotes all inputs except $\mathbf{i}$. Intuitively, the formula asserts that for every two pairs of execution traces (π_1, π_2), the value of $\mathbf{o}$ has to be the same until there is a difference between π_1 and π_2 in the input vector $\bar{\mathbf{i}}$, i.e., the inputs on which $\mathbf{o}$ may depend.

Sample hardware designs We apply RVHyper to traces generated from the following hardware designs. Note that, since RVHyper observes traces and treats the system that generates the traces as a black box, the performance of RVHyper does not depend on the size of the circuit.

Example 6 (XOR) As a first example, consider the XOR function $\mathbf{o} = \mathbf{i} \oplus \mathbf{i}'$. In the corresponding circuit, every jth output bit o_j is only influenced by the jth input bits i_j and i'_j.

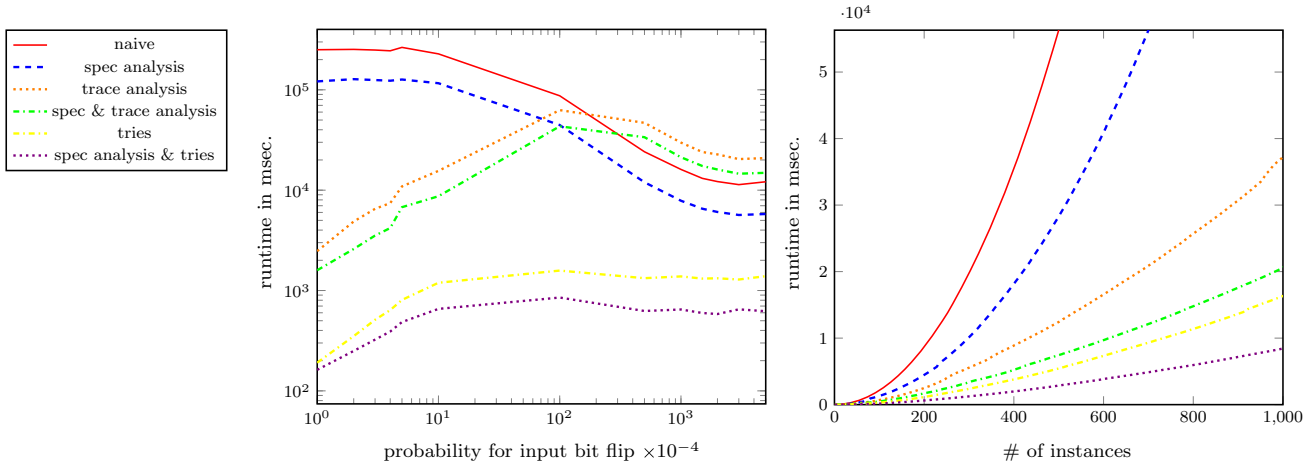


Fig. 2 Left: monitoring of black box circuits (mux example). Right: hamming-distance preserving encoder; runtime comparison of naive monitoring approach with different optimizations and a combination thereof

Table 1 Non-interference benchmark: monitored 2000 traces of length 50 with an increasing input size

Instance	Only spec analysis			Tries + spec analysis			
	# Instances	# Transitions	Time (ms)	# Instances	# Transitions	# Trie nodes	Time (ms)
8-bit	19,99,000	43,12,932	14,807	2	26,734	11,262	226
16-bit	19,99,000	27,72,001	11,166	4	34,365	87,258	285
24-bit	19,99,000	24,01,723	11,330	8	45,757	93,353	416
32-bit	19,99,000	22,36,529	13,814	16	68,364	95,237	636
40-bit	19,99,000	21,48,818	15,353	32	1,03,315	96,273	1033
48-bit	19,99,000	21,02,689	18,769	64	1,63,888	96,941	1994
56-bit	19,99,000	20,74,460	22,310	128	2,68,094	97,506	3580
64-bit	19,99,000	20,63,497	32,617	248	4,34,705	97,831	7561

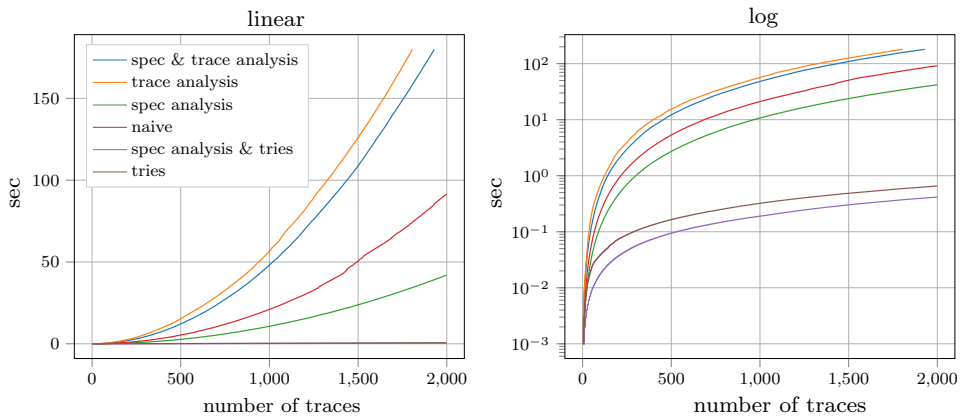


Fig. 3 Experiment of monitoring symmetry on an implementation of the Bakery protocol

Example 7 (MUX) This example circuit is depicted in Fig. 4. There is a black box combinatorial circuit, guarded by a multiplexer that selects between the two input vectors $\mathbf{i}$ and $\mathbf{i}'$ and an inverse multiplexer that forwards the output of the black box either toward $\mathbf{o}$ or $\mathbf{o}'$. Despite there being a syntactic dependency between $\mathbf{o}$ and $\mathbf{i}'$, there is no semantic dependency, i.e., the output $\mathbf{o}$ does solely depend on $\mathbf{i}$ and the selector signal.

When using the same example, but with a sequential circuit as black box, there may be information flow from the input vector $\mathbf{i}'$ to the output vector $\mathbf{o}$ because the state of the latches may depend on it. We construct such a circuit that leaks information about $\mathbf{i}'$ via its internal state.

The left part of Fig. 2 shows the total runtime of RVHyper with the different optimizations and a combination thereof. As observed in our previous experiments, the specification analysis, if applicable as in this case, is a valuable optimization consistently reducing the runtime and does so also when combined with the trace analysis. As expected, the runtime is halved by exploiting symmetry and reflexivity in the formula. From the plot, we can also infer that the trace analysis is effective in a context with a majority of redundant traces. For such a highly redundant setup, the trace analysis reduces the overall runtime of the monitoring algorithm by several magnitudes. With a decrease in similarity and redundancy in the traces, the positive effect of the trace analysis steadily decreases up until the overhead of the trace analysis itself gets noticeable. The decrease in runtime for configurations without trace analysis, which comes with reduced traces similarity, is explained by the fact that the more the input of the monitored traces is different, the earlier trace tuples can get pruned as they satisfy the specification and thereby reduce the computational burden of the algorithm. This is also the reason why the configurations with trace analysis show decreasing runtime behavior again as soon as the aforementioned effects dominate the runtime characteristics of the monitoring approach. In contrast to that, the trie optimization provides a stable improvement on the running time.

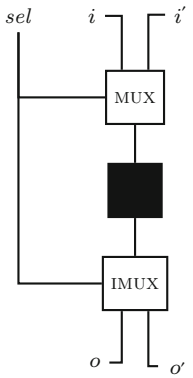


Fig. 4 MUX circuit with black box

```
1  module counter(increase,decrease,overflow);
2  input increase;
3  input decrease;
4  output overflow;
5
6  reg[2:0] counter;
7
8  assign overflow = (counter==3'b111
9   && increase && !decrease);
10
11 initial
12 begin
13   counter = 0;
14 end
15 always @($global_clock)
16 begin
17 if (increase && !decrease)
18   counter = counter + 1;
19 else if (!increase && decrease
20           && counter > 0)
21   counter = counter - 1;
22 else
23   counter = counter;
24 end
25 endmodule
```

Fig. 5 Verilog description of Example 8 (counter)

Example 8 (counter) Our last example is a binary counter with two input control bits incr and decr that increments and decrements the counter. The corresponding Verilog design is shown in Fig. 5. The counter has a single output, namely a signal that is set to one when the counter value overflows. Both inputs influence the output, but timing of the overflow depends on the number of counter bits.

Results The results of multiple random simulations are given in Table 2. Even the previous version of RVHyper was able to scale up to thousands of input traces with millions of monitor instantiations. The novel implemented optimization of RVHyper, i.e., storing the traces in a prefix tree data structure combined with our specification analysis, results in a remarkable speedup. Particularly interesting is the reduction in the number of instances in the counterexample. As there is only one input, the traces collapse in our trie data structure.

Table 2 Results of RVHyper on traces generated from circuit instances

Instance	Property	Satisfied	#Traces	Prototype		RVHyper	
				Time (ms)	#Instances	Time (ms)	#Instances
XOR	$i_0 \not\rightsquigarrow o_0$	No	18	12	222	6	18
XOR	$i_1 \not\rightsquigarrow o_0$	Yes	1000	16,913	499,500	1613	127
Counter	incr $\not\rightsquigarrow$ overflow	No	1636	28,677	1,659,446	370	2
Counter	decr $\not\rightsquigarrow$ overflow	No	1142	15,574	887,902	253	22,341
MUX	$\mathbf{i}' \not\rightsquigarrow \mathbf{o}$	Yes	1000	14,885	49,9500	496	32
MUX2	$\mathbf{i}' \not\rightsquigarrow \mathbf{o}$	No	82	140	3704	27	1913

Every instance was run ten times with different seeds, and the average is reported. Prototype refers to the first version of RVHyper [23] and RVHyper to the current implementation including the trie optimization

For the two instances, where the property is satisfied (XOR and MUX), RVHyper has not found a violation for any of the runs. For instance, where the property is violated, RVHyper was able to find counterexamples.

6 Conclusion

RVHyper monitors a running system for violations of a HyperLTL specification. We have introduced a novel trace storage optimization, based on a prefix tree data structure, to existing optimizations implemented in RVHyper.

We demonstrated the impact of the optimizations on RVHypers performance on several benchmarks of runtime verification problems. By providing a use case on how RVHyper can be used to detect spurious dependencies in hardware design, we showed how RVHyper can be used outside of classical security monitoring problems. The functionality of RVHyper thus complements model checking tools for HyperLTL, like MCHyper [26], tools for satisfiability checking, like EAHyper [22], and tools for synthesis, like BoSyHyper [21].

RVHyper is in particular useful during the development of a HyperLTL specification, where it can be used to check the HyperLTL formula on sample traces without the need for a complete model. Based on the feedback of the tool, the user can refine the HyperLTL formula until it captures the intended policy.

In our current approach, the trace analysis and the trie representation are separate optimizations that cannot be applied at the same time. The integration of the two optimization is an interesting challenge for future work.

Acknowledgements Open Access funding provided by Projekt DEAL.

References

1. Agrawal, S., Bonakdarpour, B.: Runtime verification of k-safety hyperproperties in HyperLTL. In: Proceedings of CSF, pp. 239–252. IEEE Computer Society (2016). https://doi.org/10.1109/CSF.2016.24
2. Askarov, A., Sabelfeld, A.: Tight enforcement of information-release policies for dynamic languages. In: Proceedings of CSF, pp. 43–59. IEEE Computer Society (2009). https://doi.org/10.1109/CSF.2009.22
3. Austin, T.H., Flanagan, C.: Permissive dynamic information flow analysis. In: Proceedings of PLAS, p. 3. ACM (2010). https://doi.org/10.1145/1814217.1814220
4. Bichhawat, A., Rajani, V., Garg, D., Hammer, C.: Information flow control in WebKit's JavaScript bytecode. In: Proceedings of POST. LNCS, vol. 8414, pp. 159–178. Springer (2014). https://doi.org/10.1007/978-3-642-54792-8_9
5. Bonakdarpour, B., Finkbeiner, B.: The complexity of monitoring hyperproperties. In: Proceedings of CSF, pp. 162–174. IEEE Computer Society (2018). https://doi.org/10.1109/CSF.2018.00019
6. Bonakdarpour, B., Sánchez, C., Schneider, G.: Monitoring hyperproperties by combining static analysis and runtime verification. In: Proceedings of ISoLA. LNCS, vol. 11245, pp. 8–27. Springer (2018). https://doi.org/10.1007/978-3-030-03421-4_2
7. Brett, N., Siddique, U., Bonakdarpour, B.: Rewriting-based runtime verification for alternation-free HyperLTL. In: Proceedings of TACAS. LNCS, vol. 10206, pp. 77–93 (2017). https://doi.org/10.1007/978-3-662-54580-5_5
8. Chudnov, A., Kuan, G., Naumann, D.A.: Information flow monitoring as abstract interpretation for relational logic. In: Proceedings of CSF, pp. 48–62. IEEE Computer Society (2014). https://doi.org/10.1109/CSF.2014.12
9. Clarkson, M.R., Finkbeiner, B., Koleini, M., Micinski, K.K., Rabe, M.N., Sánchez, C.: Temporal logics for hyperproperties. In: Proceedings of POST. LNCS, vol. 8414, pp. 265–284. Springer (2014). https://doi.org/10.1007/978-3-642-54792-8_15
10. Clarkson, M.R., Schneider, F.B.: Hyperproperties. J. Comput. Secur. **18**(6), 1157–1210 (2010). https://doi.org/10.3233/JCS-2009-0393
11. Coenen, N., Finkbeiner, B., Hahn, C., Hofmann, J.: The hierarchy of hyperlogics. In: Proceedings of LICS, pp. 1–13. IEEE (2019). https://doi.org/10.1109/LICS.2019.8785713
12. Coenen, N., Finkbeiner, B., Sánchez, C., Tentrup, L.: Verifying hyperliveness. In: Proceedings of CAV. LNCS, vol. 11561, pp. 121–139. Springer (2019). https://doi.org/10.1007/978-3-030-25540-4_7
13. d'Amorim, M., Rosu, G.: Efficient monitoring of omega-languages. In: Proceedings of CAV. LNCS, vol. 3576, pp. 364–378. Springer (2005). https://doi.org/10.1007/11513988_36
14. D'Argenio, P.R., Barthe, G., Biewer, S., Finkbeiner, B., Hermanns, H.: Is your software on dope? Formal analysis of surreptitiously "enhanced" programs. In: Proceedings of ESOP. LNCS, vol. 10201, pp. 83–110. Springer (2017). https://doi.org/10.1007/978-3-662-54434-1_4
15. Devriese, D., Piessens, F.: Noninterference through secure multi-execution. In: Proceedings of SP, pp. 109–124. IEEE Computer Society (2010). https://doi.org/10.1109/SP.2010.15
16. Dimitrova, R., Finkbeiner, B., Kovács, M., Rabe, M.N., Seidl, H.: Model checking information flow in reactive systems. In: Proceedings of VMCAI. LNCS, vol. 7148, pp. 169–185. Springer (2012). https://doi.org/10.1007/978-3-642-27940-9_12
17. Dimitrova, R., Finkbeiner, B., Rabe, M.N.: Monitoring temporal information flow. In: Proceedings of ISoLA. LNCS, vol. 7609, pp. 342–357. Springer (2012). https://doi.org/10.1007/978-3-642-34026-0_26
18. Duret-Lutz, A., Lewkowicz, A., Fauchille, A., Michaud, T., Renault, E., Xu, L.: Spot 2.0—a framework for LTL and ω-automata manipulation. In: Proceedings of ATVA. LNCS, vol. 9938, pp. 122–129 (2016). https://doi.org/10.1007/978-3-319-46520-3_8
19. Finkbeiner, B., Hahn, C.: Deciding hyperproperties. In: Proceedings of CONCUR. LIPIcs, vol. 59, pp. 13:1–13:14. Schloss

Dagstuhl—Leibniz-Zentrum fuer Informatik (2016). https://doi.org/10.4230/LIPIcs.CONCUR.2016.13
20. Finkbeiner, B., Hahn, C., Hans, T.: MGHyper: Checking satisfiability of HyperLTL formulas beyond the $\exists^*\forall^*$ fragment. In: Proceedings of ATVA. LNCS, vol. 11138, pp. 521–527. Springer (2018). https://doi.org/10.1007/978-3-030-01090-4_31
21. Finkbeiner, B., Hahn, C., Lukert, P., Stenger, M., Tentrup, L.: Synthesizing reactive systems from hyperproperties. In: Proceedings of CAV. LNCS, vol. 10981, pp. 289–306. Springer (2018). https://doi.org/10.1007/978-3-319-96145-3_16
22. Finkbeiner, B., Hahn, C., Stenger, M.: EAHyper: satisfiability, implication, and equivalence checking of hyperproperties. In: Proceedings of CAV. LNCS, vol. 10427, pp. 564–570. Springer (2017). https://doi.org/10.1007/978-3-319-63390-9_29
23. Finkbeiner, B., Hahn, C., Stenger, M., Tentrup, L.: RVHyper: A runtime verification tool for temporal hyperproperties. In: Proceedings of TACAS. LNCS, vol. 10806, pp. 194–200. Springer (2018). https://doi.org/10.1007/978-3-319-89963-3_11
24. Finkbeiner, B., Hahn, C., Stenger, M., Tentrup, L.: Monitoring hyperproperties. Form. Methods Syst. Des. (2019). https://doi.org/10.1007/s10703-019-00334-z
25. Finkbeiner, B., Hahn, C., Torfah, H.: Model checking quantitative hyperproperties. In: Proceedings of CAV. LNCS, vol. 10981, pp. 144–163. Springer (2018). https://doi.org/10.1007/978-3-319-96145-3_8
26. Finkbeiner, B., Rabe, M.N., Sánchez, C.: Algorithms for model checking HyperLTL and HyperCTL*. In: Proceedings of CAV. LNCS, vol. 9206, pp. 30–48. Springer (2015). https://doi.org/10.1007/978-3-319-21690-4_3
27. Guernic, G.L., Banerjee, A., Jensen, T.P., Schmidt, D.A.: Automata-based confidentiality monitoring. In: Proceedings of ASIAN. LNCS, vol. 4435, pp. 75–89. Springer (2006). https://doi.org/10.1007/978-3-540-77505-8_7
28. Hahn, C.: Algorithms for monitoring hyperproperties. In: Proceedings of Runtime Verification—19th International Conference, pp. 70–90. RV 2019, Porto, 8–11 Oct (2019). https://doi.org/10.1007/978-3-030-32079-9_5
29. Hahn, C., Stenger, M., Tentrup, L.: Constraint-based monitoring of hyperproperties. In: Proceedings of TACAS. LNCS, vol. 11428, pp. 115–131. Springer (2019). https://doi.org/10.1007/978-3-030-17465-1_7
30. Kovács, M., Seidl, H.: Runtime enforcement of information flow security in tree manipulating processes. In: Proceedings of ESSoS. LNCS, vol. 7159, pp. 46–59. Springer (2012). https://doi.org/10.1007/978-3-642-28166-2_6
31. Lamport, L.: A new solution of Dijkstra's concurrent programming problem. Commun. ACM **17**(8), 453–455 (1974). https://doi.org/10.1145/361082.361093
32. Manna, Z., Pnueli, A.: Temporal Verification of Reactive Systems: Safety. Springer, New York (1995)
33. McLean, J.: Proving noninterference and functional correctness using traces. J. Comput. Secur. **1**(1), 37–58 (1992). https://doi.org/10.3233/JCS-1992-1103
34. Roscoe, A.W.: CSP and determinism in security modelling. In: Proceedings of SP, pp. 114–127. IEEE Computer Society (1995). https://doi.org/10.1109/SECPRI.1995.398927
35. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE J. Sel. Areas Commun. **21**(1), 5–19 (2003). https://doi.org/10.1109/JSAC.2002.806121
36. Smith, G.: On the foundations of quantitative information flow. In: Proceedings of FOSSACS. LNCS, vol. 5504, pp. 288–302. Springer (2009). https://doi.org/10.1007/978-3-642-00596-1_21
37. Stucki, S., Sánchez, C., Schneider, G., Bonakdarpour, B.: Gray-box monitoring of hyperproperties. In: Proceedings of Formal Methods—the Next 30 Years—Third World Congress, pp. 406–424. FM 2019, Porto, 7–11 Oct (2019). https://doi.org/10.1007/978-3-030-30942-8_25
38. Suh, G.E., Lee, J.W., Zhang, D., Devadas, S.: Secure program execution via dynamic information flow tracking. In: Proceedings of ASPLOS, pp. 85–96. ACM (2004). https://doi.org/10.1145/1024393.1024404
39. Tabakov, D., Rozier, K.Y., Vardi, M.Y.: Optimized temporal monitors for systemC. Form. Methods Syst. Des. **41**(3), 236–268 (2012). https://doi.org/10.1007/s10703-011-0139-8
40. Vanhoef, M., Groef, W.D., Devriese, D., Piessens, F., Rezk, T.: Stateful declassification policies for event-driven programs. In: Proceedings of CSF, pp. 293–307. IEEE Computer Society (2014). https://doi.org/10.1109/CSF.2014.28
41. Yasuoka, H., Terauchi, T.: On bounding problems of quantitative information flow. In: Proceedings of ESORICS. LNCS, vol. 6345, pp. 357–372. Springer (2010). https://doi.org/10.1007/978-3-642-15497-3_22
42. Zdancewic, S., Myers, A.C.: Observational determinism for concurrent program security. In: Proceedings of CSF, p. 29. IEEE Computer Society (2003). https://doi.org/10.1109/CSFW.2003.1212703