RVHyper: A Runtime Verification Tool for Temporal Hyperproperties

Bernd Finkbeiner, Christopher Hahn, Marvin Stenger(✉), and Leander Tentrup

Reactive Systems Group, Saarland University, Saarbrücken, Germany
{finkbeiner,hahn,stenger, tentrup}@react.uni-saarland.de

Abstract. We present RVHyper, a runtime verification tool for hyperproperties. Hyperproperties, such as non-interference and observational determinism, relate multiple computation traces with each other. Specifications are given as formulas 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 counter example, in the form of a set of traces, is returned. As an example application, we show how RVHyper can be used to detect spurious dependencies in hardware designs.

1 Introduction

Hyperproperties [4] 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. HyperLTL [3] is a logic for expressing temporal hyperproperties, by extending linear-time temporal logic with *explicit* trace quantification. HyperLTL has been used to specify a variety of information-flow and security properties. Examples include classical properties like non-interference and observational determinism, as well as quantitative information-flow properties, symmetries in hardware designs, and formally verified error correcting codes [8]. While model checking and satisfiability checking tools for HyperLTL already exist [5,8], the *runtime verification* of HyperLTL specifications has so far, despite recent theoretical progress [1,2,7], not been supported by practical tool implementations.

Monitoring hyperproperties is 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. On the

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" (SFB 1223) and by the European Research Council (ERC) Grant OSARES (No. 683300).

D. Beyer and M. Huisman (Eds.): TACAS 2018, LNCS 10806, pp. 194–200, 2018.
https://doi.org/10.1007/978-3-319-89963-3_11

other hand, a runtime verification tool for hyperproperties is certainly useful, in particular 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.

In this paper, we present RVHyper, a runtime verification tool for monitoring temporal hyperproperties. RVHyper tackles this challenging problem by implementing two major optimizations: (1) a *trace analysis*, which detects all redundant traces that can be omitted during the monitoring process and (2) a *specification analysis* to detect exploitable properties of a hyperproperty, such as *symmetry*.

We have applied RVHyper in classical information-flow security, such as checking for violations of observational determinism. 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.

2 RVHyper

In this section we give an overview on the monitoring approach, including the input and output of the monitoring algorithm and the two major optimization techniques implemented in RVHyper.

Specification. The input to RVHyper is a HyperLTL specification. HyperLTL [3] is a temporal logic for specifying hyperproperties. The logic extends LTL with quantification over trace variables π and a method to link atomic propositions to specific traces. The set of trace variables is $\mathcal{V}$. Formulas in HyperLTL are given by the grammar

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

where $a \in \text{AP}$ and $\pi \in \mathcal{V}$. The finite trace semantics [2] for HyperLTL is based on the finite trace semantics of LTL. In the following, when using $\mathcal{L}(\varphi)$ we refer to the finite trace semantics of a HyperLTL formula φ. Let t be a finite trace, ϵ denotes the empty trace, and $|t|$ denotes the length of a trace. Since we are in a finite trace setting, $t[i, \ldots]$ denotes the subsequence from position i to position $|t| - 1$. Let $\Pi_{fin} : \mathcal{V} \to \Sigma^*$ be a partial function mapping trace variables to finite traces. We define $\epsilon[0]$ as the empty set. $\Pi_{fin}[i, \ldots]$ denotes the trace assignment that is equal to $\Pi_{fin}(\pi)[i, \ldots]$ for all π. We define a subsequence of t as follows.

$$t[i, j] = \begin{cases} \epsilon & \text{if } i \geq |t| \\ t[i, min(j, |t| - 1)], & \text{otherwise} \end{cases}$$

$$\begin{array}{ll}
\Pi_{fin} \vDash_T a_\pi & \text{if } a \in \Pi_{fin}(\pi)[0] \\
\Pi_{fin} \vDash_T \neg\varphi & \text{if } \Pi_{fin} \nvDash_T \varphi \\
\Pi_{fin} \vDash_T \varphi \vee \psi & \text{if } \Pi_{fin} \vDash_T \varphi \text{ or } \Pi_{fin} \vDash_T \psi \\
\Pi_{fin} \vDash_T \bigcirc\varphi & \text{if } \Pi_{fin}[1, \ldots] \vDash_T \varphi \\
\Pi_{fin} \vDash_T \varphi \,\mathcal{U}\, \psi & \text{if } \exists i \geq 0.\, \Pi_{fin}[i, \ldots] \vDash_T \psi \wedge \forall 0 \leq j < i.\, \Pi_{fin}[j, \ldots] \vDash_T \varphi \\
\Pi_{fin} \vDash_T \exists \pi.\, \varphi & \text{if there is some } t \in T \text{ such that } \Pi_{fin}[\pi \mapsto t] \vDash_T \varphi
\end{array}$$

input : $\forall^n$ HyperLTL formula φ,
set of traces T,
fresh trace t
output: satisfied or n-ary tuple
witnessing violation

$\mathcal{M}_\varphi$ = `build_template(`φ`)`;
for *each tuple* $N \in (T \cup \{t\})^n$ **do**
  if $\mathcal{M}_\varphi$ *accepts* N **then**
    proceed;
  else
    return N;
  end
end
return satisfied;

Algorithm 1. A high-level sketch of the monitoring algorithm for $\forall^n$ HyperLTL formulas.

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

$\mathcal{M}_\varphi$ = `build_template(`φ`)`
foreach $t' \in T$ **do**
  if t' *dominates* t **then**
    return T
  end
end
foreach $t' \in T$ **do**
  if t *dominates* t' **then**
    $T \coloneqq T \setminus \{t'\}$
  end
end
return $T \cup \{t\}$

Algorithm 2. Trace analysis algorithm to minimize trace storage.

For example, above mentioned observational determinism can be formalized as the HyperLTL formula $\forall\pi.\,\forall\pi'.\,(O_\pi = O_{\pi'})\,\mathcal{W}\,(I_\pi \neq I_{\pi'})$, where $\mathcal{W}$ is the weak version of $\mathcal{U}$.

Input and Output. The input of RVHyper consists of a HyperLTL formula and 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 under consideration. If RVHyper detects that the system violates the hyperproperty, it outputs a counter example, i.e, a k-ary tuple of traces, where k is the number of quantifiers in the HyperLTL formula.

Monitoring Algorithm. Given a HyperLTL formula φ and a trace set T, RVHyper processes a fresh trace under consideration as depicted in Algorithm 1. The algorithm revolves around a *monitor-template* $\mathcal{M}_\varphi$, which is constructed from the HyperLTL formula φ. The basic idea of the monitor template is that it still contains every trace variables of φ, which can be initialized with explicit traces at runtime. This way, the automaton construction of the monitor template is constructed only once as a preprocessing step.

RVHyper initializes the monitor template for each k-ary combination of traces in $T \cup \{t\}$. If one tuple violates the hyperproperty, RVHyper returns that k-ary tuple of traces as a counter example, otherwise RVHyper returns *satisfied*.

Trace Analysis: Minimizing Trace Storage. The main obstacle in monitoring hyperproperties is the potentially unbounded space consumption. RVHyper

uses a *trace analysis* 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 counter example.

RVHyper's trace analysis is based on the definition of trace redundancy: we say a fresh trace t is (T, φ)-redundant, if T is a model of φ if and only if $T \cup \{t\}$ is a model of φ. The idea, depicted in Algorithm 2, is to check if another trace t' contains at least as much informations as t: we say a t' dominates t if $\bigwedge_{\pi \in \mathcal{V}} \mathcal{L}(\mathcal{M}_\varphi[t'/\pi]) \subseteq \mathcal{L}(\mathcal{M}_\varphi[t/\pi])$. For a fresh incoming trace, RVHyper performs this language inclusion check in both directions in order to compute the minimal trace set that must be stored to monitor the hyperproperty under consideration.

Specification Analysis: Decreasing Running Time. RVHyper uses a *specification analysis*, which is a preprocessing step that analyzes the HyperLTL formula under consideration. RVHyper detects 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 [7].

Symmetry is especially interesting because 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 pair of traces which violates the symmetry condition: $\exists \pi. \exists \pi'. \big((O_\pi = O_{\pi'}) \, \mathcal{W} \, (I_\pi \neq I_{\pi'})\big) \not\leftrightarrow \big((O_{\pi'} = O_\pi) \mathcal{W} (I_{\pi'} \neq I_\pi)\big)$. If the resulting formula turns out to be unsatisfiable, RVHyper omits the symmetric instantiations of the monitor automaton, which turns out to be, especially in combination with RVHypers *trace analysis*, a major optimization in practice [7].

Implementation. RVHyper[1] is written in C++. It uses *spot* for building the deterministic monitor automata and the *Buddy* BDD library for handling symbolic constraints. We use the HyperLTL satisfiability solver EAHyper [5,6] to determine whether the input formula is reflexive, symmetric, or transitive. Depending on those results, we omit redundant tuples in the monitoring algorithm.

3 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.

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

Input and Output. The input to RVHyper is a set of traces where the propositions match the atomic propositions of the 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 counter example.

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

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

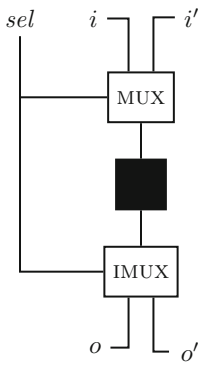

Fig. 1. MUX circuit with black box

where $\overline{\boldsymbol{i}}$ denotes all inputs except $\boldsymbol{i}$. Intuitively, the formula asserts that for every two pairs of execution traces (π_1, π_2) the value of $\boldsymbol{o}$ has to be the same until there is a difference between π_1 and π_2 in the input vector $\overline{\boldsymbol{i}}$, i.e., the inputs on which $\boldsymbol{o}$ 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 1 (*XOR*).* As a first example, consider the XOR function $\boldsymbol{o} = \boldsymbol{i} \oplus \boldsymbol{i}'$. In the corresponding circuit, every j-th output bit o_j is only influenced by the j-the input bits i_j and i'_j.

*Example 2 (*MUX*).* This example circuit is depicted in Fig. 1. There is a black box combinatorial circuit, guarded by a multiplexer that selects between the two input vectors $\boldsymbol{i}$ and $\boldsymbol{i}'$ and an inverse multiplexer that forwards the output of the black box either towards $\boldsymbol{o}$ or $\boldsymbol{o}'$. Despite there being a syntactic dependency between $\boldsymbol{o}$ and $\boldsymbol{i}'$, there is no semantic dependency, i.e., the output $\boldsymbol{o}$ does solely depend on $\boldsymbol{i}$ 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 $\boldsymbol{i}'$ to the output vector $\boldsymbol{o}$ because the state of the latches may depend on it. We construct such a circuit that leaks information about $\boldsymbol{i}'$ via its internal state.

*Example 3 (*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. 2. 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.

```
module counter(increase,
  decrease, overflow);
input increase;
input decrease;
output overflow;

reg[2:0] counter;

assign overflow = (counter
  == 3'b111 && increase
  && !decrease);


initial
begin
  counter = 0;
end
always @($global_clock)
begin
if (increase && !decrease)
  counter = counter + 1;
else if (!increase && decrease
          && counter > 0)
  counter = counter - 1;
else
  counter = counter;
end
endmodule
```

Fig. 2. Verilog description of Example 3 (counter).

Table 1. Results of RVHyper on traces generated from circuit instances. Every instance was run 10 times with different seeds and the average is reported.

Instance	Property	Satisfied	# traces	Length	Time	# instances
XOR	$i_0 \not\rightsquigarrow o_0$	no	18	5	12 ms	222
XOR	$i_1 \not\rightsquigarrow o_0$	yes	1000	5	16913 ms	499500
counter	$incr \not\rightsquigarrow overflow$	no	1636	20	28677 ms	1659446
counter	$decr \not\rightsquigarrow overflow$	no	1142	20	15574 ms	887902
MUX	$\boldsymbol{i'} \not\rightsquigarrow \boldsymbol{o}$	yes	1000	5	14885 ms	499500
MUX2	$\boldsymbol{i'} \not\rightsquigarrow \boldsymbol{o}$	no	82	5	140 ms	3704

Results. The results of multiple random simulations are given in Table 1. Despite the high complexity of the monitoring problem, RVHyper is able to scale up to thousands of input traces with millions of monitor instantiations (cf. Algorithm 1). RVHyper's optimizations, i.e., keeping only a minimal set of traces and reducing the number of instances by the specification analysis, are a key factor to those results. For the two instances where the property is satisfied (XOR and MUX), RVHyper has not found a violation for any of the runs. For instances where the property is violated, RVHyper is able to find counter examples. While counter examples can be found quickly for XOR and MUX2, the counter instances need more traces since the chance of finding a violating pair of traces is lower.

4 Conclusion

RVHyper monitors a running system for violations of a HyperLTL specification. The functionality of RVHyper thus complements model checking tools for HyperLTL, like MCHyper [8], and tools for satisfiability checking, like EAHyper [6]. 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.

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)
2. Brett, N., Siddique, U., Bonakdarpour, B.: Rewriting-based runtime verification for alternation-free HyperLTL. In: Legay, A., Margaria, T. (eds.) TACAS 2017. LNCS, vol. 10206, pp. 77–93. Springer, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54580-5_5
3. Clarkson, M.R., Finkbeiner, B., Koleini, M., Micinski, K.K., Rabe, M.N., Sánchez, C.: Temporal logics for hyperproperties. In: Abadi, M., Kremer, S. (eds.) POST 2014. LNCS, vol. 8414, pp. 265–284. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54792-8_15
4. Clarkson, M.R., Schneider, F.B.: Hyperproperties. J. Comput. Secur. **18**(6), 1157–1210 (2010)
5. Finkbeiner, B., Hahn, C.: Deciding hyperproperties. In: Proceedings of CONCUR, LIPIcs, vol. 59, pp. 13:1–13:14. Leibniz-Zentrum fuer Informatik (2016)
6. Finkbeiner, B., Hahn, C., Stenger, M.: EAHyper: satisfiability, implication, and equivalence checking of hyperproperties. In: Majumdar, R., Kunčak, V. (eds.) CAV 2017. LNCS, vol. 10427, pp. 564–570. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-63390-9_29
7. Finkbeiner, B., Hahn, C., Stenger, M., Tentrup, L.: Monitoring hyperproperties. In: Lahiri, S., Reger, G. (eds.) RV 2017. LNCS, vol. 10548, pp. 190–207. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67531-2_12
8. Finkbeiner, B., Rabe, M.N., Sánchez, C.: Algorithms for model checking HyperLTL and HyperCTL*. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9206, pp. 30–48. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21690-4_3