Monitoring Hyperproperties by Combining Static Analysis and Runtime Verification

Borzoo Bonakdarpour[1], Cesar Sanchez[2], and Gerardo Schneider[3]

[1] McMaster University, Canada, borzoo@mcmaster.ca
[2] IMDEA Software Institute, Spain, cesar.sanchez@imdea.org
[3] University of Gothenburg, Sweden gerardo@cse.gu.se

Abstract. Hyperproperties are properties whose reasoning involve sets of traces. Examples of hyperproperties include information-flow security properties, properties of coding/decoding systems, linearizability and other consistency criteria as well as privacy properties like data minimality. We study the problem of runtime verification of hyperproperties expressed as HyperLTL formulas that involve quantifier alternation. We first show that even for a simple class of temporal formulas, virtually no $\forall\exists$ property can be monitored, independently of the observations performed. To manage this problem, we propose to use a combination of static analysis with runtime verification. By using static analysis/verification, one typically obtains a model of the system that allows to limit the source of "hypothetical" traces to a sound over-approximation of the traces of the system. This idea allows to extend the effective monitorability of hyperproperties to a larger class of systems and properties. We exhibit some examples where instances of this idea have been exploited, and discuss general applications of the principle. A second contribution of this paper is the idea of departing from the convention that all traces come from executions of a single system. We show cases where traces are extracted by the observed traces of agents, by projections of a single global trace, or by executions of different (but related) programs.

1 Introduction

In this paper, we study how to monitor hyperproperties [17], in particular $\forall\exists$ and $\exists\forall$ fragments of the temporal logic HyperLTL [16]. A *monitor* is a piece of software that observes and analyzes execution traces of a said *monitored system* under analysis. In particular, we are concerned with *runtime verification*, that aims to determine whether the given monitored program satisfies a pre-defined property expressed in a behavioral specification language. Monitors may analyze the executions of the program *online* (while the program is running) or *offline* (the execution traces of the program are collected and analyzed a-posteriori)[4]. In runtime verification, the monitor is generated automatically from the property,

[4] The definition of offline monitoring also includes when traces are obtained from other sources than the program running in its real environment (e.g., in a simulation environment, or traces not coming from the real program but from a model for instance).

written in a formal language, usually as a logical formula or as an automaton. As in testing, runtime verification may be *black box* or *white box*, depending on whether the monitor can only observe the behavior of the system (the input-ouput events or states), or whether the monitor has additionally access to the internals of the monitored program.

In the context of trace logics (where properties can be evaluated independently on each individual trace) it is known that not all properties are *monitorable*, because the monitors can only observe finite prefixes of executions. Monitorability means that there exits a monitor that can declare correctly whether the property is permanently satisfied or violated in all future extensions of the observed prefix traces. Previous results in trace logics show how to build monitors for some combinations of *safety* and *co-safety* properties. Formalisms proposed for generating monitors include LTL adapted for finite paths [8, 21, 26], regular expressions [40], rule based languages [6], rewriting [37], streams [19] and automata [18].

In this paper we study a richer class of properties called *hyperproperties* [17], that is, properties that are defined over a set of sets of traces instead of over set of traces. Many security properties, like *information-flow* properties such as *non-interference*, are hyperproperties. In particular, many of the common and interesting security properties are *2-safety* hyperproperties [17], meaning that it can be expressed using universal quantification over pairs of finite traces. Monitoring hyperproperties requires to collect and reason about multiple finite runs, and not only a single run under scrutiny. As with trace logics, there is a source of uncertainty because the traces under observation are finite and correspond to a prefix of execution. However, in monitoring hyperproperties there is an additional source of imperfect observability, because if the collection of traces is approximated dynamically by the traces executed, it is possible that the set contains also traces that have not observed.

It has been shown that it is possible to construct monitors for $\forall\forall$ hyperproperties if the temporal formula is restricted to safety [1, 10, 24], where some limitations of monitorability are also explored. In particular, one cannot, in general, determine at runtime whether a $\forall\forall$ property is satisfied because this may require to obtain all possible executions of the system. However, one can monitor $\forall\forall$ safety properties for violations.

In this paper, we show that $\forall\exists$ (and $\exists\,\forall$) are not monitorable in general. Then, we explore new ideas on how to monitor properties of the form $\forall\exists$ by using a *grey box* approach, that is by exploiting *certain* information we have about the system to perform the monitoring task. This additional information can be present as a model or specification of the system under analysis, which soundly approximates the set of runs of the actual system. We identified at least two different cases on how to use such models depending on their nature: (i) when the model is the formal specification of the ideal (expected) behavior of the program; (ii) when the model is extracted statically from the given program. These models can be used as a kind of *oracle* to instantiate one of the quantifiers of the specified property. Consider, for example, the goal of detecting violations of a $\forall\exists$ property.

The first step is to complement the property and detect satisfactions of the $\exists\forall$ dual property. Then, at runtime, one can use the observed trace as a prefix of a witness for the outermost quantifier (the $\exists$ quantifier), and use the model to then safely check whether all traces satisfy the temporal relational formula. If the model indeed over-approximates the actual set of traces of the system, this approach can monitor violations of the $\forall\exists$ property. Then, the approach we propose reduces the monitoring activity to efficiently check the model at runtime. We exhibit in this paper examples where bounded model checking could be used, and others where symbolic execution and SAT solvers may be used. We also discuss directions for developing solutions with general applicability.

Finally, we depart from the standard view of hyperproperties, where traces that can instantiate the quantifiers are traditionally taken from the set of executions of the very same program (that is, from the program being monitored). Here, we generalize this view allowing different quantifiers to be instantiated with traces from different "trace sets". One particular case is the monitoring approaches sketched above where one set of traces is taken from the execution of the system under observation, while the other set of traces are runs of the model. Other instances are possible, and we discuss some in this paper. For example, if different quantifiers can be instantiated with the observations of different process, nodes or agents, many properties of concurrent and distributed systems (e.g., mutual exclusion) can be viewed as hyperproperties. We argue that this interpretation makes sense in practice, as the traces coming from the different (distributed or concurrent) processes are not always available at the same time to compute a global trace view, or they are simply not available because we only have access to the local information of a given process.

The rest of the paper is organized as follows. Section 2 presents the challenges faced when monitoring hyperproperties, particularly when $\forall\exists$ properties are considered. Section 3 presents examples of hyperproperties and how the approach of instantiating quantifiers with different trace sets can lead to hyperproperties. Section 4 elaborates on the idea of combining static and dynamic analysis for monitoring two concrete hyperproperties, namely, linearizability and data minimization. Section 5 outlines potential techniques to monitor more general fragments of HyperLTL formulas. Finally, Section 6 presents related work and Section 7 concludes.

2 The Challenge of Monitoring Hyperproperties

In this section, we discuss the notion of monitorability in runtime verification of trace properties and of hyperproperties.

2.1 Monitoring Trace Properties

We first revisit monitorability of trace properties. We focus on LTL [28, 34] as a representative language. Let u be a finite execution trace and φ be an LTL formula. If all infinite traces extending u satisfy φ, then we say that u

permanently or *inevitably satisfies* φ. Similarly, if all infinite extensions of u violate φ, then we say that u *permanently violates* φ. Monitoring consists on declaring, whether u permanently satisfies or permanently violates φ, or none. The latter occurs when there exists a future extension to u that satisfies φ and another future extension of u that violates φ. For instance, consider the LTL formula $\varphi = a \,\mathbf{U}\, b$ and the finite trace $u = aaa$. This trace can be extended to trace $u' = aaab$, which permanently satisfies φ, or can be extended to $u'' = aaa(\neg a \wedge \neg b)$, which permanently violates φ.

Pnueli and Zaks [35] characterize an LTL formula φ as *monitorable* for a finite trace u, if u can be extended to another finite trace that can be evaluated as a satisfying or violating execution with respect to φ at run time. We call the extension $u\sigma$ of an observation u a *trace extension*. For example, the LTL formula $\Diamond p$ is monitorable, since every finite trace u can be extended to one of the form $u \cdots p \cdots$, i.e., a finite trace in which p has become true. On the contrary, formula $\Box\Diamond p$ is not monitorable, because there is no way to tell at run time whether or not in the future p will be visited infinitely often.

The above discussion clarifies the challenges in RV for LTL formulas: monitoring an LTL formula boils down to determining the verdict of the formula for a finite trace with an eye on the possible future extensions. Consequently, a system can be monitored in a *black box* manner with respect to a rich class of LTL formulas. In LTL monitoring, the monitor only needs to observe a single evolving trace of the system without having access to the code. In the next subsection, we show that this is not the case for hyperproperties.

2.2 Monitoring Hyperproperties

We focus now on HyperLTL [16] as a representative language for hyperproperties. Consider the HyperLTL formula:

$$\varphi_1 = \forall\pi.\forall\pi'.\Box(a_\pi \leftrightarrow a_{\pi'})$$

Intuitively, this formula requires that for any instantiation of trace variables π and π', say to concrete traces u and u', the value of proposition a in the i-th position of u should agree with the value of a in the i-th position of u'. For example, let $u = (\neg a)(\neg a)(a)(\neg a)(a)$ and $u' = (\neg a)(a)(a)(\neg a)(a)$. This pair of finite traces permanently violates φ because u and u' do not agree on a in the second position of the traces. Thus, if a HyperLTL formula φ that is only universally quantified and the inner LTL formula is monitorable for violations (for traces of pairs of states) then φ can be monitored for violations. Declaring satisfaction for such a formula requires, in principle, examining *all* traces, which essentially becomes infeasible at run time. Even if the monitor could decide that for all trace extensions of pairs of traces seen, the inner property holds, it cannot know whether offending pairs of traces can be potentially emitted by the system. Dually, for an existential HyperLTL formula only satisfaction can be detected (and the inner formula is required to be monitorable for satisfaction).

Consider now the following HyperLTL formula with one quantifier alternation

$$\varphi_2 = \forall\pi.\exists\pi'.\Box(a_\pi \leftrightarrow \neg a_{\pi'})$$

and consider the same traces $u = (\neg a)(\neg a)(a)(\neg a)(a)$ and $u' = (\neg a)(a)(a)(\neg a)(a)$. These traces do not satisfy the inner temporal formula, that is (u, u') permanently violates $\square(a_\pi \leftrightarrow \neg a_{\pi'})$. However, this fact alone is not a witness for a violation of φ_2. For instance, a trace $u'' = (a)(a)(\neg a)(a)(\neg a)$ is (at least as a prefix) a perfect witness for $\exists\pi'$. A formula like φ_2 can never be declared permanently satisfied or violated, simply because at run time, it is not possible to tell whether for *all* traces, there exists another trace that satisfies the inner LTL formula. Thus, in addition to challenges of LTL monitoring, HyperLTL monitoring involves reasoning about quantified traces and future extensions not just in length, but also the observed set of traces. Agrawal and Bonakdarpour [1] defined monitorability of HyperLTL as follows:

> A HyperLTL formula φ is *monitorable* for violation if every finite set U of finite traces can be extended to another finite set of finite traces U' (both by extending traces in U and by adding new traces) that guarantees that every extension of U' violates φ.

The definition for monitorability for satisfaction is analogous. An important limitation of this definition (and the work in [1]) is that the technique is restricted to the *alternation-free* fragment, that is, to formulas of the form $\forall\forall$ or formulas of the form $\exists\exists$. Extending this notion of monitorability is infeasible in general for $\forall\exists$ properties, even for the simplest temporal properties. We sketch now a very general result about the impossibility of monitoring $\forall\exists$ hyperproperties.

Consider a (relational) state predicate $P(x, x')$ on two copies of variables. The intuition of such a predicate P is to represent a relational invariant between the corresponding states in two traces. Without loss of generality, assume that P is such that for every valuation u there is a valuation u' that makes $P(u, u')$ true. Also, assume that if one plugs the same valuation v to x and x', the predicate $P(v, v)$ is false (we call such P an irreflexive relational predicate). This latter constraint is not a restriction as every predicate P can be extended into $(P \wedge (b \leftrightarrow \neg b'))$ for a fresh Boolean variable b. We claim that the following formula is not monitoriable

$$\psi = \forall\pi\exists\pi'.\square P(x_\pi, x_{\pi'})$$

Consider an arbitrary finite observation U, collected by the monitor. We show that U has a model U_{good} (set of infinite traces) that violates ψ and an extensionU_{bad} that satisfies ψ. Since the monitor cannot distinguish which of the two the system can generate, the monitor cannot declare a conclusive verdict.

- We first show that U can be extended to a counterexample U_{bad} that violates ψ. Assume that all traces in U have the same length (otherwise simply add arbitrary padding states to the shorter traces). Then, create U' by adding the same state a extending every observed trace in U. Since $P(a, a)$ is false, every pair of traces in U falsifies $\square P(x_\pi, x_{\pi'})$. The set U_{bad} is obtained by extending U' to infinite traces arbitrarily. This shows that ψ cannot be monitored for satisfaction because every observation U can be extended to a counterexample.

– We show now that U can be extended to a model U_{good} that satisfies ψ, by taking the universal set $U_{\mathsf{good}} = \Sigma^\omega$ (the set of all infinite traces). It is easy to see that the universal set satisfies ψ as every trace σ has a corresponding trace σ' that satisfies $\square P(x_\pi, x_{\pi'})$. One can simply make, for all positions i in the trace, $\sigma'(i)$ the assignment to the state variables that make $P(\sigma(i), \sigma'(i)$ true. This shows that ψ cannot be monitored for violations because very observation U can be extended to a model.

Since ψ cannot be monitored for violation or for satisfaction, ψ is not monitorable.

The above discussion illustrates a challenge in RV for HyperLTL formulas: monitoring a HyperLTL formula boils down to determining the verdict of the formula for a finite set of finite traces with an eye on the future extensions in both size and length. Consequently, HyperLTL monitoring cannot be implemented as a *black box* technique for virtually all formulas with quantifier alternations. Thus, we advocate for the development of *grey box* techniques, where the monitor observes a set of traces at run time, and can use some static information about the system under observation to narrow down the class of plausible systems (in order to be able to determine the verdict). In this paper, we argue that such approximation is possible by using static analysis/verification techniques.

3 Examples of Hyperproperties

We show here examples of hyperproperties, some of which are $\forall\forall$, others are $\forall\exists$, and one $\forall\forall\exists$. Some of these hyperproperties follow the "standard" viewpoint in which traces are alternative executions of a single system under observation (Section 3.1). Others, however, are obtained by taking a different point of view (Section 3.2): different quantifiers can be instantiated with traces from different systems, or from different components of a running system (e.g., the traces of execution of two threads).

3.1 Classic Examples

In this subsection, we present requirements that are inherently hyperproperties regardless of how execution traces are collected.

Information-flow Security Information-flow security properties stipulate how information may propagate from inputs to outputs. Such policies may belong to alternation-free as well as alternating HyperLTL formulas. For example, let the observable input to a system be the atomic proposition i and the output the atomic proposition o. Then, *observational determinism* can then be expressed as the following alternation-free HyperLTL formula:

$$\varphi_{\mathsf{obs}} = \forall\pi.\ \forall\pi'.\ \square\,(i_\pi \leftrightarrow i_{\pi'}) \rightarrow \square\,(o_\pi \leftrightarrow o_{\pi'}),$$

This formula establishes that if two traces π and π' agree globally on i, then they must also globally agree on o. Following the discussion in Section 2, formula φ_{obs} can be monitored to detect violations, but not satisfaction. On the contrary, Goguen and Meseguer's *noninterference* stipulates that, for all traces, the low-observable output must not change when all high inputs are removed:

$$\varphi_{\mathsf{gmni}} = \forall\pi.\exists\pi'.(\square\lambda_{\pi'}) \wedge \square(o_\pi \leftrightarrow o_{\pi'})$$

where $\lambda_{\pi'}$ expresses that all of the secret inputs in the current state of π' have dummy value λ, and o denotes publicly observable output propositions. GMNI is clearly an alternating formula and following the discussion in Section 2 cannot be monitored using a blackbox technique. We refer the reader to [9] for examples of security properties that result in HyperLTL formulas with more quantifier alternations.

Linearizability Informally, linearizability is a consistency model for concurrent data structures and distributed transactions and establishes that every concurrent execution of a given datatype is *observationally equivalent* to a *sequential execution* of the same datatype (or of a specified datatype like a list, set, stack, etc). Here, observationally equivalent means that all values returned by methods of the datatype when executed concurrently are the same values returned by some sequential execution of the same client behavior. Similarly, sequential execution means that each method invocation is run to completion atomically and uninterruptedly once the internal execution of the method starts. Formally, linearizability is a hyperproperty of the form:

$$\varphi_{\mathsf{lz}} = \forall\pi.\exists\pi'.Seq(\pi') \wedge Obs(\pi,\pi') \tag{1}$$

The first observation in (1) is that for trace variables π and π' to be observationally equivalent (as denoted by $Obs(\pi,\pi')$) the instructions that threads execute outside an invocation to the datatype or internally within the body of the dataype are not visible. Only the execution of calls and returns are visible, and $Obs(\pi,\pi')$ claim that these events are identical in both traces.

3.2 Hyperproperties Obtained by Different Source of Traces

Properties of Concurrent and Distributed Systems Consider the well-known *mutual exclusion* requirement, where two processes cannot be in the critical section at the same time. This requirement is a safety trace property and can be expressed in LTL by the following formula:

$$\square(\neg cs_1 \vee \neg cs_2),$$

where cs_i for $i \in \{1,2\}$ indicates that process i is in the critical section. This formula expresses mutual exclusion over the global states of the system. However, in many concurrent and distributed systems, traces are collected from individual

processes or computing cores because in some circumstances it is not easy or even possible to construct interleaved traces. This way, it is more convenient to express a requirement such as mutual exclusion as a hyperproperty where traces correspond to local executions. Thus, mutual exclusion can be expressed in HyperLTL by formula

$$\varphi_{\mathsf{me}} = \forall\pi.\forall\pi'.\Box(\neg cs_\pi \vee \neg cs_{\pi'}),$$

where π and π' are traces collected from two different processes, cores, or threads.

A similar example is *data races*. A data race happens when there are two memory accesses (at least one being a write) performed concurrently by two threads, to the same location and that are not protected by synchronization operations. For a given memory location, this property can be expressed as a $\forall\forall$ property where traces are the local executions of the threads.

Some examples of $\forall\exists$ properties in the setting of concurrent and distributed systems include the following:

- *Egalitarian schedulers.* Some problems in distributed systems can only be solved by breaking symmetries, like for example to exploit in a predictable way that process identifiers are ordered. However, one may wish that the system does not penalize a process unnecessarily (which for example could cause starvation). One way to express that a scheduler or resource manager does not unnecessarily favor a process over another is to express that for all excutions (of the system) there is an alternative execution (of the same system), where the actions taken by the scheduler to brake symmetries are reversed.
- *Reads preceded by writes.* For any trace with a read on a given register, there must exist a trace with a write on that register performed in a previous moment.
- *Resource waiting.* If a process is blocked waiting for a resource, then there must exist another process that acquired that resource in the past and still holds the resource at the time the first process is waiting for it.

Data Minimization We present now another instance of alternating hyperproperties from a different context, namely privacy. According to the Article 5 of the *General Data Protection Regulation* proposal (GDPR) "Personal data must be [...] limited to what is necessary in relation to the purposes for which they are processed" [23].[5] The above is usually called the *data minimization* principle. Though data minimization is about both the collection and the processing of data, we are here only concerned with the former. In particular, we are taking the point of view of Antignac et. al. [2, 3], where the concept of *data minimizer* has been defined as a pre-processor that filters the input of the given program in such a way that the functionality of the program does not change but the

[5] The *General Data Protection Regulation* (EU —2016/679) was adopted on 27 April 2016, and it will enter into application 25 May 2018.

program only receives data that is necessary and sufficient for the intended computation. From there they derived the concept of data minimization and they showed how to obtain data minimizers for both the *monolithic* case (only one source of input) and the *distributed* case (more than one, independent, source of inputs). Note that the setting is for deterministic (functional) programs, and the different definitions are based only on the observable behavior relating inputs and outputs.

Results concerning the monitoring of violations of data minimization (i.e., *non-minimality*) for the monolithic case and the so-called *strong* distributed minimality has been studied in [32, 33], so we will only focus here in case of ("weak") distributed minimality as this is a $\forall\forall\exists$ hyperproperty.

We now give a formal definition of distributed minimality as a hyperproperty. The definition is essentially equivalent to the one in [32] but rephrased to be consistent with the temporal style of HyperLTL[6]. We compare (terminating) funcions with n parameters. We view a run of the funcion as a sequence of states, where the state predicate *end* denotes that the trace reaches the function final state and halts. In order to extend terminating executions to infinite traces, we repeat the halting state ad infinitum. We consider the state variables $in_1, \ldots, in_n$ to represent the inputs to the function, and the variable o which represents the output, assigned once the function has been computed. We introduce the following auxiliary predicates

$$\mathit{output}(\pi, \pi') : \Diamond\Big(\mathit{end}(\pi) \wedge \mathit{end}(\pi') \wedge o(\pi) = o(\pi')\Big)$$

The predicate *output* establishes that the functions represented by the two traces terminate, and that the computed output is the same. Similarly, we define

$$\mathit{differ}_j(\pi, \pi') \ : \ in_j(\pi) \neq in_j(\pi') \qquad\qquad \mathit{almost}_j(\pi, \pi') \ : \ \bigwedge_{k \neq j} in_k(\pi) = in_k(\pi')$$

The predicate differ_j establishes that the inputs to the function are different in the parameter j, and almost_j that the input to the function agree on all parameters except possibly on j. Now we can define distributed minimality for input j as:

$$\varphi^j_{\mathsf{dm}} \ : \ \forall\pi.\forall\pi'.\exists\pi''. \begin{pmatrix} \mathit{output}(\pi, \pi') \\ \wedge \\ \mathit{differ}_j(\pi, \pi') \end{pmatrix} \rightarrow \begin{pmatrix} \mathit{almost}_j(\pi, \pi'') \\ \wedge \\ \neg \mathit{output}(\pi, \pi'') \end{pmatrix}$$

and finially distributed minimality as

$$\varphi_{\mathsf{dm}} \ : \ \bigwedge_{j=1..n} \varphi^j_{\mathsf{dm}}.$$

[6] A stronger version of distributed minimality, which is a $\forall\forall$ hyperproperty, is given in [33].

Example 1 (From [32]). Consider the function $\text{OR} : \mathbb{B} \times \mathbb{B} \to \mathbb{B}$ function, which was shown not to be monolithic minimal [2, 3]. This function is distributed-minimal. We have two input sources I_1 and I_2 both of sort $\mathbb{B}$. For the first input source, for each possible pair of distinct values in that position (that is $(0, \_)$ and $(1, \_)$), we can find satisfactory input tuples yielding different results (e.g., $((0,0),(1,0))$ since $\text{OR}(0,0) \neq \text{OR}(1,0)$).[7] Similarly, for input source 2, for each possible pair of distinct values in that position ($(\_, 0)$ and $(\_, 1)$), we have that the tuples $(0,0)$ and $(0,1)$ satisfy the definition ($\text{OR}(0,0) \neq \text{OR}(1,0)$). □

Distributed non-minimality is simply the negation of the above formula, and thus a $\exists\exists\forall$ formula. As previously discussed, none of these properties (distributed minimality and its negation) are monitorable in a black-box fashion.

4 Monitoring with the Aid of Static Verification. Examples

We present two practical case studies that exploit static information to perform monitoring of specific hyperproperties.

4.1 Monitoring Linearizability

We propose the following combination of static and runtime verification to monitor linearizability violations:

1. First, the code of the concurrent datatype is *statically verified* to satisfy the pre-post specification of the programming abstraction that the datatype is meant to implement, under the assumption of sequential invocations. This activity can be performed with mature deductive verification techniques for sequential programs, using for example the KeY infrastructure for Java programs.
2. Then, at *runtime*, a *monitor* receives events about calls and returns from concurrent clients that exercise the concurrent datatype. This monitor exhaustively explores the set of possible sequences of atomic executions (*using the specification*) that are observationally equivalent to the visible events in the observed concurrent trace.

We illustrate this approach with a concrete example, the Trieber stack [41]. It is easy to prove using state-of-the-art deductive verification that the code in Fig. 1—when executed sequentially—implements a stack. Then, at runtime, the monitor we propose works as follows.

1. The monitor maintains a set of possible states that the datatype may be into. In our example, the state will be concrete stacks. Additionally, the state of the monitor also contains two sets:

[7] Note that the pair $((0,1),(1,1))$ would not satisfy the definition, but this is fine as the definition only requires that at least one such tuple exists.

```
void push (Item e) {
  Node * new_hd = new Node(e);
  Node * hd;
  do {
     hd = top.get();
     new_hd->next = hd;
  }
  while (!CAS(top, hd, new_hd);
}
```

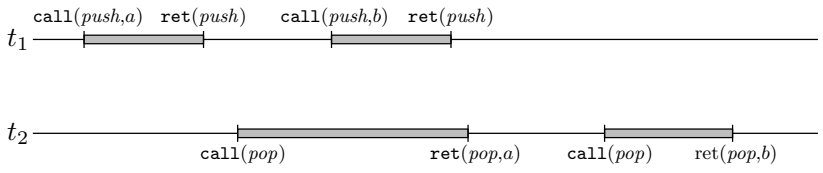
```
Item pop () {
     Node * hd;
     Node * new_hd;
     do {
        hd = top.get();
        if (hd == null) {
           return null;
        }
        new_hd = hd.next;
     }
     while (!CAS(top, hd, new_hd);
     return hd.item;
}
```

Fig. 1. Trieber stack implementation

   - Pending operations of the form $(t, f, args)$, where t is a thread identifier, f the name of the method that t is executing and *args* is the arguments to the method;
   - Operations effectively executed but not yet returned, stored as (t, f, val) where t and f are as above, and *val* is the returned value.

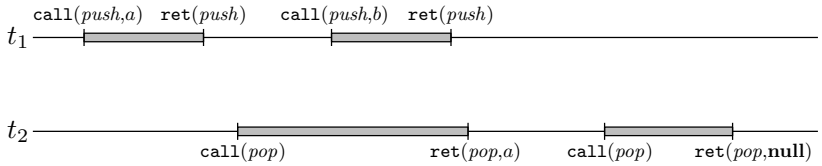
   Each tuple $(State, Pending, Executed)$ is a *plausible state* of the monitor, where *State* represents the state of the stack, *Pending* represents the pending operations, and *Executed* the operations that were performed effectively on the object but whose return have not been observed yet.
2. The initial state of the monitor is the empty stack with no pending or executed operations. This is the only plausible initial state.
3. When an operation invocation occurs in the observed trace, denoted by the execution of the instruction that calls the method from the calling thread, *every* plausible state maintained by the monitor is extended by adding the invoked operation to its pending set.
4. When an operation finishes in the observed trace, as observed by the return instruction being executed, the monitor computes—for each plausible state—the following possible successors. A successor consists on executing, in some order, a subset of the pending operations followed by the observed operation. The output of the observed operation must be equal to the observed value, otherwise the successor does not correspond to the observed outcome and it is removed from the plausible set.

If, at some point the set of plausible states is empty, then there is a violation of linearizability. For example, consider the following execution:



The state of the monitor after the event "call($push$,b)" executed by thread t_1 will be $\langle (a), \{(t_2, pop), (t_1, push, b)\}, \emptyset \rangle$. This state reflects that the stack contains only a and that there are two pending operations. After the observable event "ret($push$)" is executed, the monitor computes the following plausible states: $\langle (b : a), (t_2, pop), \emptyset \rangle$ and $\langle (b), \emptyset, (t_2, pop, a) \rangle$ Only later, when the pending pop operation is observed to return an a, the first plausible state is descarded by the monitor because executing the pop from state $(b : a)$ would return b and not a.

Consider now an incorrect implementation of the `CAS` (compare-and-swap) operation that is non-atomic. In this case, the following execution could be generated (hint: thread t_2 is preempted in the middle of the `CAS` operation):



In this case, the monitor would raise a violation of linearizability as none of the two plausible states can lead to the event "ret(pop,**null**)" by t_2.

4.2 Monitoring Data Minimization

We now present a second example based on detecting violations of data minimization. In [2, 3] Antignac et. al. provided a white-box approach to statically synthesize a minimizer for the different notions of data minimization. The problem of finding a minimizer is undecidable in general, and thus the approach only works for very specific cases. In a nutshell, the approach consists in extracting the symbolic execution tree of the program under consideration and applying a SAT solver to the constraints given by such a symbolic tree by making a conjunction with the different possible outputs. Unfortunately, the approach does not scale. First, the symbolic execution is in general a coarse over-approximation of the real behavior unless the user provides more precise annotations in the form of pre- and post-conditions and invariants. Second, if the output domain is infinite or it is too big, it is not feasible to expect the SAT solver to terminate in a reasonable time.

We now sketch how this approach, apart from being interesting from a theoretical perspective can be integrated into a runtime environment in order to increase the possibility of detecting violations to minimality via monitoring. Let us assume a program $\mathcal{P}$ which for the sake of simplicity of presentation we assume has three different inputs from different sources, that is $\mathcal{P} : I_1 \times I_2 \times I_3 \rightarrow O$. We also assume that we have a model of $\mathcal{P}$ in the form of its symbolic tree and that we have access to a SAT solver which is called with the program's symbolic tree as well as concrete inputs and outputs obtained from observed executions. We call $\mathcal{M}$ the module executing the solver over the symbolic tree and the input/output pairs given at run time. Roughly speaking, our approach would work as follows:

1. After each execution of $\mathcal{P}$, $\mathcal{M}$ gets the input/output pair $\langle (i_1, i_2, i_3), o \rangle$.
2. $\mathcal{M}$ (symbolically) executes such a pair and puts it as belonging to a given partition characterizing the inputs for the first input giving the same output.
3. Whenever an execution with a different value for the first input parameter is found but with the same output as a previous execution, then $\mathcal{M}$ is called in order to find out whether there are values for the other two input parameters giving a different output. If this is the case, then nothing can be said yet. If such values cannot be found, then distributed minimization is violated.

The above process is in fact executed for each input (this could be done in parallel) as the definition is symmetric with respect to all the input entries.

5 Towards Covering More General HyperLTL Fragments

We now sketch a direction to systematically generalize the previous examples to a wider range of properties and systems.

5.1 Predictive Semantics for LTL

Employing static analysis and verification techniques in order to enhance RV has been investigated before. For example, in [43], static analysis is introduced to allow *predictive semantics* to the 3-valued LTL, where LTL formulas are evaluated not only with respect to a runtime finite trace, but also by assistance from an abstract model of the system under inspection. The predictive semantics aim at anticipating whether the satisfaction or violation of the specification is inevitable in all continuations of the trace for the giving abstract model, even when the observed execution by itself does not imply the inevitable verdict for all trace extensions. More specifically, let u be a finite trace and P be a program under inspection with respect to an LTL formula φ. Recall that the 3-valued semantics of LTL prescribe that if any future extension of u satisfies (respectively, violates) φ, then u *permanently* satisfies (respectively, violates) φ. Realistically, the future extensions of interest should only be the possible continuations of u that P can generate. One efficient way to reason about all future extensions of u in P is to compute an over-approximate abstract model of P (denoted $\widehat{P}$) and then check, given u, whether or not for all infinite extensions σ of u in the trace set of $\widehat{P}$, $u\sigma \models \varphi$ holds. If it is indeed the case that $u\sigma \models \varphi$, then we are guaranteed that u permanently satisfies φ, as the trace set of $\widehat{P}$ subsumes the trace set of P. Similarly, if for all extensions σ, we have $u\sigma \not\models \varphi$, then we can conclude that u permanently violates φ.

5.2 The $\exists\forall$ and $\forall\exists$ Fragments

As discussed in Section 2, most non-trivial alternating HyperLTL formulas are non-monitorable, unless we are able to have additional information about the sytem (white- or grey-box approach). We now sketch a potential grey-box approach inspired by the predictive semantics of LTL to monitor HyperLTL formulas. Let P be a program, $\varphi = \exists\pi.\forall\pi'.\psi$ be a HyperLTL formula, and u be a

finite execution trace obtained at run time. In order to monitor P with respect to φ, it is sufficient to answer the following decision problem:

Is there an extension σ of u, such that for all executions τ of P, $(u\sigma, \tau) \models \psi$?	$(*)$

One possibility is to use a model-checker to perform the necessary state exploration. Intuitively, the model to be verified is the cross product of u with an over-approximate abstract model $\widehat{P}$ in a way similar to self-composition [7][8]. Let us denote this new model by P'. We also have to modify ψ to reflect what really needs to be verified in P'. Let us denote this formula bv ψ'. In summary, this approach would require algorithms that generate P' and ψ', such that $P' \models \psi'$ if and only if u is permanently satisfied (or permanently violated).

We sketch here an alternative approach using bounded model checking [15] to decide $(*)$, for a given u for a class of HyperLTL formulas. Consider a property $\varphi = \exists\pi.\forall\pi'.\psi(\pi, \pi')$ where ψ is a *co-safety* formula, for example

$$\varphi = \exists\pi.\forall\pi'.\Diamond(a_\pi \wedge a_{\pi'}).$$

First, observe that since ψ is a co-safety property, if a prefix (u, v) satisfies ψ, then all extensions of (u, v) will satisfy ψ as well. Then, given an over-approximation $\widehat{P}$ of P, bounded model checking (BMC) can be used to unroll $\widehat{P}$ and compute a product of u with all the traces of $\widehat{P}$ of length $|u|$. If the BMC instance declares that there is no counterexample of length $|u|$, then there will be no extension that becomes a counterexample of ψ. Hence, the formula φ will be permanently satisfied and u is indeed a witness of the outermost existential quantifier. If, on the other hand, BMC produces a counterexample there are three scenarios:

1. The observation u is too short to serve as a witness for the existential quantifier. In particular, there may be extensions of u that satisfy the property φ and can be checked by the method described above.
2. The model $\widehat{P}$ has not been explored to sufficent depth to prove a satisfaction of the co-safety property ψ.
3. The approximated model $\widehat{P}$ is too coarse and includes spurious traces.

One research challenge for this approach is to explore to detect spurious counterexamples and how to refine $\widehat{P}$ to continue the exploration. Another challenge is to refine the BMC procedure to effectively compute the set of offending traces of length $|u|$ and design efficient methods to exploit this information to successive BMC queries (for lengths $|u| + 1$, etc), thus creating an incremental approach.

Note that if the HyperLTL formula is of the form $\forall\pi.\exists\pi'.\psi(\pi, \pi')$, where ψ is a *safety* formula, then the same procedure can be used to detect violations. One such formula is $\forall\pi.\exists\pi'.\Box(p_\pi \leftrightarrow p_{\pi'})$. To monitor this formula, we can compute the negation $\neg\varphi = \exists\pi.\forall\pi'.\neg\psi$. This formula is now covered by the previous case,

[8] We speculate that the abstract model $\widehat{P}$ may be computed using different techniques, e.g., predicate abstraction, symbolic execution, etc.

as $\neg\psi$ is a co-safety formula. Given a runtime finite trace u, if BMC reports satisfaction, then we can conclude that φ is permanently violated by any extension of the observed trace.

5.3 More Efficient Monitoring of the $\exists\exists$ and $\forall\forall$ Fragments

Finally, our suggested grey-box approach can also be applied to improve the monitoring procedure for the alternation-free fragments of HyperLTL considered in [1, 10, 24]. Consider formula $\varphi = \exists\pi.\exists\pi'.\psi$, where ψ is a co-safety formula. Similar to the $\exists\forall$ fragment explained above, given a runtime finite trace u, one can (1) instantiate π with u, and (2) obtain a model P' by composing u with $\widehat{P}$. The difference with the $\exists\forall$ procedure before is in handling $\exists\pi'.\psi'$. Unlike in the $\exists\forall$ case, here, we verify whether $P' \models \neg\psi'$. If this results in a counterexample, this counterexample is a positive witness to π' and, hence, the formula φ is permanently satisfied. Analogously, we can monitor for violation of formulas of the form $\forall\pi.\forall\pi'.\psi$, where ψ' is a safety formula.

6 Related Work

Monitoring Hyperproperties The notion of hyperproperties was introduced by Clarkson and Schneider [17]. HyperLTL [16] is a temporal logic for hyperproperties. Model checking algorithms for HyperLTL were introduced in [25]. Runtime verification algorithms for HyperLTL include both automata-based algorithms [1,24] and rewriting-based algorithms [10]. These RV approaches either work for alternation-free formulas or for alternating formulas only if the size of the trace set for monitoring does not grow. Also, the monitoring technique only considers the traces seen, whereas in our grey-box approach, the monitor may detect errors by exploring traces of the model that have not been seen.

Static Analysis Sabelfeld et al. [39] survey the literature focusing on static program analysis for enforcement of security policies. In some cases, for example just-in-time compilation techniques and dynamic inclusion of code at runtime in web browsers, static analysis does not guarantee secure execution at runtime. Type systems, frameworks for JavaScript [13] and ML [36] are some approaches to monitor information flow. Several tools [22, 29, 30] add extensions such as statically checked information flow annotations to the Java language. Clark et al. [14] present verification of information flow for deterministic interactive programs. In [4], Assaf and Naumann propose a technique for designing runtime monitors based on an abstract interpretation of the system under inspection.

Dynamic analysis Russo et al. [38] concentrate on permissive techniques for the enforcement of information flow under flow-sensitivity. It has been shown that in the flow-insensitive case, a sound purely dynamic monitor is more permissive than static analysis. However, they show the impossibility of such a

monitor in the flow-sensitive case. A framework for inlining dynamic information flow monitors has been presented by Magazinius et al. [27]. The approach by Chudnov et al. [12] uses hybrid analysis instead and argues that due to JIT compilation processes, it is no longer possible to mediate every data and control flow event of the native code. They leverage the results of Russo et al. [38] by inlining the security monitors. Chudnov et al. [11] again use hybrid analysis of 2-safety hyperproperties in relational logic. Austin and Flanagan [5] implement a purely dynamic monitor, however, restrictions such as "no-sensitive upgrade" were placed. Some techniques deploy taint tracking and labelling of data variables dynamically [31, 44]. Zdancewic et al. [42] verify information flow for concurrent programs.

SME Secure multi-execution [20] is a technique to enforce non-interference. In SME, one executes a program multiple times, once for each security level, using special rules for I/O operations. Outputs are only produced in the execution linked to their security level. Inputs are replaced by default inputs except in executions linked to their security level or higher. Input side effects are supported by making higher-security-level executions reuse inputs obtained in lower-security-level threads. This approach is sound only for deterministic languages.

7 Conclusion

We presented in this paper preliminary work on how to monitor hyperproperties that are not monitorable in general when considering a black-box approach. In particular, we considered hyperproperties with one quantifier alternation ($\forall\exists$ and $\forall\forall\exists$) for which we give an informal argument about their non-monitorability. We provided initial ideas on how to monitor a large class of formulas in the fragment for violation, by monitoring the negation of such a fragment for satisfaction. Our proposal is based on a suitable combination of static analysis/verification techniques with runtime verification, thus taking a grey-box approach. Additionally, our techniques consider traces coming not only from the monitored system but also from other sources. In particular, we have explored the use of a specification of the system as a trace generator for the inner universal quantifier, as well as a model obtained from the system via symbolic execution and predicate abstraction.

The main idea behind our approach is that the current real execution of the system accounts for the outermost quantifier, while a static analysis/verification is applied to explore "runs" of the model accounting for the innermost quantifier. Note that we do not claim anticipation as we might need more traces or more observations before we could get a final verdict.

Besides the above, we have departed from the view of considering global traces of concurrent and distributed systems, and consider local traces instead. In this way, many of the traditional properties of such systems can be casted as hyperproperties (e.g., mutual exclusion and data races). Some of such properties are in the fragment $\forall\forall$ while others are of the form $\forall\exists$.

Our main research agenda now is to generalize the approach and apply it to all the cases presented in this paper while identifying other interesting hyperproperties fitting in the fragment under consideration.

References

1. S. Agrawal and B. Bonakdarpour. Runtime verification of k-safety hyperproperties in HyperLTL. In *CSF'16*, pages 239–252, 2016.
2. T. Antignac, D. Sands, and G. Schneider. Data minimisation: A language-based approach (long version). Technical Report abs/1611.05642, CoRR–arXiv.org, 2016.
3. T. Antignac, D. Sands, and G. Schneider. Data Minimisation: A Language-Based Approach. In *IFIP SEC'17*, volume 502 of *IFIP Advances in Information and Communication Technology (AICT)*, pages 442–456, 2017.
4. M. Assaf and D. A. Naumann. Calculational design of information flow monitors. In *CSF'16*, pages 210–224, 2016.
5. T. H. Austin and C. Flanagan. Efficient purely-dynamic information flow analysis. In *ACM Trans. on Programming Languages and Systems*, pages 113–124, 2009.
6. H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In *Proc. of VMCAI'04*, LNCS 2937, pages 44–57. Springer, 2004.
7. G. Barthe, P. R. D'Argenio, and T. Rezk. Secure information flow by self-composition. In *CSFW'04*, pages 100–114. IEEE Computer Society Press, 2004.
8. A. Bauer, M. Leucker, and C. Schallhart. Runtime verification for LTL and TLTL. *ACM T. Softw. Eng. Meth.*, 20(4):14, 2011.
9. B. Bonakdarpour and B. Finkbeiner. The complexity of monitoring hyperproperties. In *CSF'18*, 2018. To appear.
10. N. Brett, U. Siddique, and B. Bonakdarpour. Rewriting-based runtime verification for alternation-free HyperLTL. In *TACAS'17*, pages 77–93, 2017.
11. A. Chudnov, G. Kuan, and D. A. Naumann. Information flow monitoring as abstract interpretation for relational logic. In *CSF'14*, pages 48–62, 2014.
12. A. Chudnov and D. A. Naumann. Information flow monitor inlining. In *Proceedings of CSF*, pages 200–214, 2010.
13. R. Chugh, J. A. Meister, R. Jhala, and S. Lerner. Staged information flow for javascript. In *Proceedings of PLDI*, pages 50–62, 2009.
14. D. Clark and S. Hunt. Non-interference for deterministic interactive programs. In *Proceedings of Formal Aspects in Security and Trust*, pages 50–66, 2008.
15. E. M. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.
16. M. R. Clarkson, B. Finkbeiner, M. Koleini, K. K. Micinski, M. N. Rabe, and C. Sánchez. Temporal logics for hyperproperties. In *POST'14*, volume 8414 of *LNCS*, pages 265–284. Springer, 2014.
17. M. R. Clarkson and F. B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010.
18. C. Colombo, G. J. Pace, and G. Schneider. LARVA — Safer Monitoring of Real-Time Java Programs (Tool Paper). In *SEFM'09*, pages 33–37. IEEE Computer Society, 2009.
19. B. D'Angelo, S. Sankaranarayanan, C. Sánchez, W. Robinson, B. Finkbeiner, H. B. Sipma, S. Mehrotra, and Z. Manna. LOLA: Runtime monitoring of synchronous systems. In *TIME'05*, pages 166–174. IEEE CS Press, 2005.

20. D. Devriese and F. Piessens. Noninterference through secure multi-execution. In *31st IEEE Symposium on Security and Privacy, S&P*, pages 109–124, 2010.
21. C. Eisner, D. Fisman, J. Havlicek, Y. Lustig, A. McIsaac, and D. V. Campenhout. Reasoning with temporal logic on truncated paths. In *Proc. of CAV'03*, volume 2725 of *LNCS 2725*, pages 27–39. Springer, 2003.
22. W. Enck, P. Gilbert, S. Han, V. Tendulkar, B. Chun, P. L. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Trans. Comput. Syst.*, 2014.
23. European Commission. Proposal for a Regulation of the European Parliament and of the Council on the protection of individuals with regard to the processing of personal data and on the free movement of such data (GDPR). Technical Report 2012/0011 (COD), European Commission, January 2012.
24. B. Finkbeiner, C. Hahn, M. Stenger, and L. Tentrup. Monitoring hyperproperties. In *RV'17*, volume 10548 of *LNCS*, pages 190–207. Springer, 2017.
25. B. Finkbeiner, M. N. Rabe, and C. Sánchez. Algorithms for model checking HyperLTL and HyperCTL*. In *CAV'15*, pages 30–48, 2015.
26. K. Havelund and G. Roşu. Synthesizing monitors for safety properties. In *Proc. of TACAS'02*, LNCS 2280, pages 342–356. Springer, 2002.
27. J. Magazinius, A. Russo, and A. Sabelfeld. On-the-fly inlining of dynamic security monitors. *Computers & Security*, 31(7):827–843, 2012.
28. Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems.* Springer-Verlag, 1995.
29. A. C. Myers. Jflow: Practical mostly-static information flow control. In *POPL'99*, pages 228–241, 1999.
30. A. C. Myers and B. Liskov. Complete, safe information flow with decentralized labels, 1998.
31. S. Nair, P. N. D. Simpson, B. Crispo, and A. S. Tanenbaum. A virtual machine based information flow control system for policy enforcement. *ENTCS*, 197(1):3–16, 2008.
32. S. Pinisetty, T. Antignac, D. Sands, and G. Schneider. Monitoring data minimisation. Technical Report abs/1801.02484, CoRR–arXiv.org, 2018.
33. S. Pinisetty, D. Sands, and G. Schneider. Runtime Verification of Hyperproperties for Deterministic Programs. In *FormaliSE'18*. ACM, 2018. To appear.
34. A. Pnueli. The temporal logic of programs. In *FOCS'77*, pages 46–67. IEEE Computer Society Press, 1977.
35. A. Pnueli and A. Zaks. PSL Model Checking and Run-Time Verification via Testers. In *14th Int. Symp. on Formal Methods (FM)*, pages 573–586, 2006.
36. F. Pottier and V. Simonet. Information flow inference for ml. In *POPL'02*, pages 319–330, 2002.
37. G. Roşu and K. Havelund. Rewriting-based techniques for runtime verification. *Automated Software Engineering*, 12(2):151–197, 2005.
38. A. Russo and A. Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *CSF'10*, pages 186–199, 2010.
39. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.
40. K. Sen and G. Roşu. Generating optimal monitors for extended regular expressions. *ENTCS*, 89(2):226–245, 2003.
41. R. K. Treiber. Systems programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center, April 1986.
42. S. Zdancewic and A. C. Myers. Observational determinism for concurrent program security. In *Computer Security Foundations Workshop*, pages 29–, 2003.

43. X. Zhang, M. Leucker, and W. Dong. Runtime verification with predictive semantics. In *NASA FM'12*, pages 418–432, 2012.
44. Y. Zhu, J. Jung, D. Song, T. Kohno, and D. Wetherall. Privacy scope: A precise information flow tracking system for finding application leaks. Technical report, EECS Department, University of California, Berkeley, Oct 2009.