Formal Methods in System Design
https://doi.org/10.1007/s10703-020-00358-w

Gray-box monitoring of hyperproperties with an application to privacy

Sandro Stucki[1] · **César Sánchez**[2] · **Gerardo Schneider**[1] · **Borzoo Bonakdarpour**[3]

Received: 28 February 2020 / Accepted: 9 December 2020
© The Author(s) 2021

Abstract
Runtime verification is a complementary approach to testing, model checking and other static verification techniques to verify software properties. *Monitorability* characterizes what can be verified (monitored) at run time. Different definitions of monitorability have been given both for trace properties and for *hyperproperties* (properties defined over sets of traces), but these definitions usually cover only some aspects of what is important when characterizing the notion of monitorability. The first contribution of this paper is a refinement of classic notions of monitorability both for trace properties and hyperproperties, taking into account, among other things, the computability of the monitor. A second contribution of our work is to show that *black-box* monitoring of HyperLTL (a logic for hyperproperties) is in general unfeasible, and to suggest a *gray-box* approach in which we combine static and runtime verification. The main idea is to call a static verifier as an oracle at run time allowing, in some cases, to give a final verdict for properties that are considered to be non-monitorable under a black-box approach. Our third contribution is the instantiation of this solution to a privacy property called *distributed data minimization* which cannot be verified using black-box runtime verification. We use an SMT-based static verifier as an oracle at run time. We have implemented our gray-box approach for monitoring data minimization into the proof-of-concept tool *Minion*. We describe the tool and apply it to a few case studies to show its feasibility.

Keywords Runtime verification · Monitorability · Hyperproperty · LTL · HyperLTL · Data minimization · Security · Privacy

1 Introduction

Imagine yourself organizing an international conference on formal methods with parallel tracks spread over multiple conference venues. While the caterers prepare beverages and

---

This research has been partially supported by the United States NSF SaTC Award 1813388, by the Swedish Research Council (*Vetenskapsrådet*) under Grant 2015-04154 “PolUser”, by the Madrid Regional Government under Project S2018/TCS-4339 “BLOQUES-CM”, by EU H2020 Project 731535 “Elastest”, and by Spanish National Project PGC2018-102210-B-100 “BOSCO”.

Extended author information available on the last page of the article

Published online: 02 February 2021

snacks for the early morning coffee break on the first day of the conference, you find yourself pondering the following questions:

1. Will there be enough coffee for the participants of all the different tracks during the upcoming coffee break?
2. Will the coffee at the different venues be served simultaneously?
3. If one of the venues runs out of coffee, will there still be coffee at one of the other venues?

Questions like these, involving different possible behaviors of local state across multiple parts of a complex system, are called *hyperproperties*. The above questions can be compactly formalized in the following three HyperLTL formulas:

$$(1)\ \forall\pi.\,\Box ☕_{\pi} \qquad (2)\ \forall\pi.\forall\pi'.\,\Box(☕_{\pi} \rightarrow ☕_{\pi'}) \qquad (3)\ \forall\pi.\exists\pi'.\,\Box(\neg ☕_{\pi} \rightarrow ☕_{\pi'})$$

where π and π' denote conference venues.

In this paper, we investigate the *monitorability* of such hyperproperties. Typically, one considers a property to be monitorable if it is possible to reach a scenario (within a finite number of steps) which definitely satisfies or violates the property. For example, we can easily imagine future scenarios where questions 1 and 2 have definitive answers. When any of the conference venues run out of coffee during a break, we have detected a violation of (1). Even if the empty coffee dispensers are later replenished, the property, being a safety hyperproperty, remains *permanently violated*. Similarly, any pair of venues where one runs out of coffee before the other constitutes a permanent violation of (2). Thus, monitorability is easy to establish for (1) and (2). Monitorability of (3), on the other hand, is surprisingly subtle. According to earlier work on the topic [3], the property should *not* be monitorable; yet we can build a monitor for (3) in practice. This apparent contradiction and its resolution are the subject of the first half of this paper.

Monitoring hyperproperties is of interest beyond the coffee breaks of academics meetings. Most notably, hyperproperties arise naturally when characterizing the *security* and *privacy* of information systems. Indeed, monitoring systems for security and privacy violations at run time is one of the main motivations behind our work. As a case in point, we apply our insights on monitoring hyperproperties to a particular privacy hyperproperty called *distributed data minimality* (DDM) in the second half of this paper.

Beyond this particular application, our theoretical results apply to the broader context of *runtime verification* (RV). Runtime verification is a computing analysis technique based on observing executions of a system to check its expected behavior against predefined properties. When using RV, one first generates a monitor from a specification, ideally automatically, and then uses the monitor to analyze the behavior of a system under study. RV is considered to be “a practical application of formal verification” and “a less *ad-hoc* approach complementing conventional testing and debugging” [40]. Unlike static formal verification, RV sacrifices completeness since it can only analyze (a finite collection of) finite traces of a system under observation, which are typically (a finite set of) prefixes of (a potentially infinite set of) potentially unbounded computations. Despite this apparent limitation, there are properties that can only be verified at run time. See [25,30,40] for surveys on RV, and the recent book [9].

When applying static verification techniques to complex systems, one is forced to consider decidability issues. Similar considerations apply to runtime verification, where the properties of interest are those that are *monitorable*. Monitorability was first defined by Pnueli and Zaks as follows. A property expressed in LTL is monitorable (after observing a prefix trace u) if there is an extension of u that would violate or satisfy the property [38]. We call this notion *semantic black-box monitorability*. It is *semantic* because this notion of monitorability

defines a decision problem (the existence of a satisfying or violating trace extension) without requiring a corresponding decision procedure. It is *black-box* because this definition only considers the property without further information about the program/system being monitored, so every extended observation is possible and must be considered. As initial research in RV originated in the model checking community, it was natural to consider the problem of monitoring LTL formulae. The problem of monitoring LTL (for similar notions of monitorability) is quite well-studied [13,26,38] (see also [10]). For other settings, though, these "classical" definitions are not very suitable: a property may be semantically monitorable even though no algorithm to monitor that property exists.

A hyperproperty is a property defined over sets of traces (i.e. a hyperproperty defines a set of sets of traces). Therefore, monitoring hyperproperties implies reasoning about multiple traces simultaneously (seminal work on monitoring hyperproperties include [3,17,24]). Most security properties are hyperproperties [18], including confidentiality, integrity, non-interference, non-inference, etc. This is also the case for privacy properties, such as *data minimization* [7,36]. The notion of monitorability for hyperproperties in [3] extends the original definition by not only considering whether extensions of an observed trace would violate or satisfy the property, but also considering extensions of the set of observed traces. A big limitation of the existing notions of monitorability in the literature is that they completely ignore the role of the system being monitored.

In this paper, we consider a more fine-grained concept of monitorability providing a more comprehensive landscape of different aspects to be considered along the three dimensions of the cube depicted in Fig. 1. The first dimension of the cube expresses that the monitor can either reason about single traces, or about multiple traces simultaneously (the trace/hyper dimension). The second dimension is concerned with how much we know about the system being monitored (the black/white dimension). If we have full knowledge of the system and its analysis is completely precise, we call this *white-box* monitoring. *Black-box* monitoring refers to the classic approach of assuming zero knowledge about the system and crafting general monitors that provide sound verdicts for every system. White- and black-box monitoring are the extreme ends of a spectrum. Between them, we find various degrees of *gray-box* monitoring, in which the monitor uses *some* information about the system (approximate sets of executions). This partial knowledge may be given by a model of the system in addition to the observed finite execution. Note that we may or may not have access to the source code of the system being monitored; what is important is to have a model of it. The third dimension (computability) considers the computability limitations of the monitors themselves as programs, that is, it focuses on the computational power of the monitors.

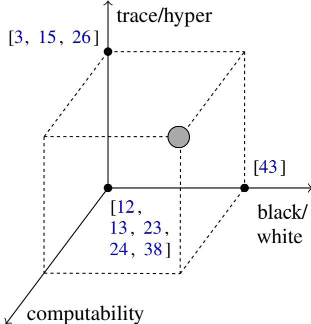

Fig. 1 The monitorability cube

We prove in this paper that a large class of hyperproperties that involve quantifier alternations are not back-box monitorable in general. To work around this discouraging result, we propose a gray-box approach based on a combination of static and runtime verification that allows us to still give a definitive verdict (violation or satisfaction) for certain properties that are not black-box monitorable. In particular, our approach uses static verification over a model of the system as an oracle at run time: given the set of already observed (finite) traces, the oracle is used to try to reach a verdict concerning further (not yet seen) traces.

We then apply our approach to a specific privacy property called *data minimality*, more specifically *distributed data minimality* (DDM). The principle of data minimization (defined in Article 5 of the EU General Data Protection Regulation [21]) stipulates that only data that is (semantically) used by a program should be collected and processed. When data is collected from independent sources, the principle is called distributed data minimization. DDM can be expressed as a formula in HyperLTL with one quantifier alternation, of the form $\forall\forall\exists\exists\varphi$. It has been shown in [36] that a stronger version of DDM is black-box monitorable for violations of the property, but nothing is said about DDM, though it is left implicit that it is non-monitorable ("formulas with alternating quantifiers are not monitorable in general").

Our approach to monitor violations of DDM is as follows. We create a gray-box monitor that dynamically observes and collects traces for the negation of DDM ($\exists\exists\forall\forall\neg\varphi$), which are then considered to be potential witnesses for the existential part. The monitor then invokes an oracle (a model extracted from the source code in the form of its symbolic execution tree, on which we use an SMT solver) to soundly decide the universally quantified inner sub-formula. Our approach is sound but the monitor may give an inconclusive answer, depending on the precision of the static verifier. We present a proof-of-concept gray-box monitor for DDM called *Minion*[1] and we apply it to a few case studies to show its feasibility.

This paper is a revised and extended version of a paper presented at the 23rd International Symposium on Formal Methods (FM'19) [41]. Besides including more detailed examples, and full proofs of our theoretical results, we have added a new section describing our tool *Minion* in more detail, and new theoretical results as explained below. In summary, the contributions of this paper are:

(1) A generalized version of HyperLTL parametrized over relational structures, with a richer, more expressive core logic that is better suited to reasoning about security hyperproperties. (Sect. 2.)
(2) A unified semantic framework of monitorability abstracting over any particular choice of specification logic. By instantiating our framework to LTL or HyperLTL, we obtain the familiar notions of monitorability for trace- and hyperproperties (Sect. 2).
(3) A novel and richer definition of monitorability that, besides the choice of specification logic, also considers different degrees of information about the system being monitored (gray-box monitoring), and the computational power of the monitor (computability) (Sect. 3).
(4) A method for gray-box monitoring by enhancing runtime verification with oracles that uses static analysis and static verification in order to enable the possibility of monitoring properties that are non-monitorable in a black-box manner (Sect. 3).
(5) We express DDM as a hyperproperty and study its monitorability using our new gray-box approach. In particular, the oracle is based on symbolic execution and SMT solvers (Sect. 4).
(6) We describe a proof-of-concept implementation of our gray-box monitor for DDM, apply it to some representative examples, and present an empirical evaluation (Sect. 5).

[1] Freely available online at https://github.com/sstucki/minion/.

We start by recapitulating LTL and HyperLTL and summarizing existing notions of monitorability. Comparison with related work and our conclusions are presented in the last two sections of the paper.

2 Background

Let Σ be a set, called the *alphabet*. We call each element of Σ a *letter* (or an *event*). Throughout the paper, Σ^ω denotes the set of all infinite sequences (called *traces*) over Σ, and Σ^* denotes the set of all finite traces over Σ. For a trace $t \in \Sigma^\omega$ (or $t \in \Sigma^*$), $t[i]$ denotes the ith element of t, where $i \in \mathbb{N}$. We use $|t|$ to denote the length (finite or infinite) of trace t. Also, $t[i, j]$ denotes the subtrace of t from position i up to and including position j (or ϵ if $i > j$ or if $i > |t|$). In this manner $t[0, i]$ denotes the prefix of t up to and including i and $t[i, ..]$ denotes the suffix of t from i (including i).

Given a set X, we use $\mathcal{P}(X)$ for the set of subsets of X and $\mathcal{P}_{\mathit{fin}}(X)$ for the set of finite subsets of X. Let u be a finite trace and t a finite or infinite trace. We denote the concatenation of u and t by ut. Also, $u \preceq t$ denotes the fact that u is a prefix of t. Given a finite set U of finite traces and an arbitrary set W of finite or infinite traces, we say that W extends U (written $U \preceq W$) if, for all $u \in U$, there is a $v \in W$, such that $u \preceq v$. Note that every trace in U is extended by some trace in W (we call these *trace extensions*), and that W may also contain additional traces with no prefix in U (we call these *set extensions*).

2.1 LTL and relational HyperLTL

We now briefly introduce LTL and HyperLTL. Let AP be a finite set of *atomic propositions* and define the alphabet Σ as $\Sigma = 2^{\mathsf{AP}}$. The syntax of LTL [37] is:

$$\varphi ::= a \mid \neg\varphi \mid \varphi \vee \varphi \mid \bigcirc\varphi \mid \varphi \,\mathsf{U}\, \varphi$$

where $a \in \mathsf{AP}$. The semantics of LTL is given by associating to a formula the set of traces $t \in \Sigma^\omega$ that it accepts:

$$\begin{array}{lll} t \models a & \text{iff} & a \in t[0] \\ t \models \neg\varphi & \text{iff} & t \not\models \varphi \\ t \models \varphi_1 \vee \varphi_2 & \text{iff} & t \models \varphi_1 \text{ or } t \models \varphi_2 \\ t \models \bigcirc\varphi & \text{iff} & t[1, ..] \models \varphi \\ t \models \varphi_1 \,\mathsf{U}\, \varphi_2 & \text{iff} & \text{for some } i,\ t[i, ..] \models \varphi_2 \text{ and for all } j < i,\ t[j, ..] \models \varphi_1 \end{array}$$

We will also use the usual derived operators ($\Diamond\varphi \equiv \mathit{true}\,\mathsf{U}\,\varphi$) and ($\Box\varphi \equiv \neg\Diamond\neg\varphi$).

All properties expressible in LTL are *trace properties* (each individual trace satisfies the property or not, independently of any other trace). Some important properties, such as information-flow security policies (including confidentiality, integrity and secrecy), cannot be expressed as trace properties but require reasoning about two (or more) independent executions (perhaps from different inputs) simultaneously. Such properties are called *hyperproperties* [18]. HyperLTL [19] is a temporal logic for hyperproperties that extends LTL by allowing explicit quantification over execution traces.

We present here a generalized version of HyperLTL with a richer core logic than those commonly found in the literature. Whereas HyperLTL is typically defined with a propositional core logic similar to that of LTL, here we allow formulas to use atomic *relation symbols* of non-zero arity. This extension is motivated by relational properties from security and privacy,

which involve comparisons of values ranging over unbounded I/O domains across multiple executions of a system. For example, the well-known *non-interference* property may be expressed symbolically as

$$\forall \pi_1 . \forall \pi_2 . \Big(\mathrm{in}(\pi_1) =_L \mathrm{in}(\pi_2) \rightarrow \mathrm{out}(\pi_1) =_L \mathrm{out}(\pi_2) \Big).$$

Non-interference is clearly a hyperproperty and, as such, has been a target for run-time verification via HyperLTL along with other, similar security properties [3,36]. Yet, statements like $\mathrm{in}(\pi_1) =_L \mathrm{in}(\pi_2)$ cannot be expressed in propositional logic in general and are therefore, strictly speaking, beyond the scope of HyperLTL. The above is true unless $\mathrm{in}(\pi)$ and $\mathrm{out}(\pi)$ range over finite I/O domains. That said, even if realistic systems operate on finite data in practice, encoding a property such as non-interference as a propositional formula is impractical for large/unbounded I/O domains (text, video, etc.). To overcome this limitation, we strengthen the core of HyperLTL to a predicate logic parametrized by a given σ-*structure*.

Definition 1 (*signatures and structures*) A *(relational) signature* σ is a pair $\sigma = (S, \mathrm{ar})$, where S is a set of *relation symbols* and $\mathrm{ar}\colon S \rightarrow \mathbb{N}$ assigns an *arity* $\mathrm{ar}(r)$ to each $r \in S$. A σ-*structure* is a pair (A, I), where A is a set, called the *domain*, and I is an *interpretation function* assigning to each $r \in S$ an $\mathrm{ar}(r)$-ary relation $I(r) \subseteq A^{\mathrm{ar}(r)}$. A *(relational) structure* $\mathcal{A}$ is a triple $\mathcal{A} = (\sigma_{\mathcal{A}}, A_{\mathcal{A}}, I_{\mathcal{A}})$ consisting of a relational signature $\sigma_{\mathcal{A}}$ and its interpretation.

For simplicity, we restrict ourselves to single-sorted relational structures (that is, the set A is the domain interpreting the only sort), which is sufficient for the purpose of this paper. However, the definition of Relational HyperLTL given below can easily be adapted to many-sorted structures with standard techniques. Similarly, our results can be easily extended to cover function symbols. We leave these extensions and a thorough discussion of the logical and proof-theoretic properties of HyperLTL over a structure $\mathcal{A}$ for future work.

Let $\mathcal{V}_o$ be a finite set of *object variables* and $\mathcal{V}_t$ a countably infinite set of *trace variables*. Object variables $x, y, z, \ldots \in \mathcal{V}_o$ denote the *observables* of a system, e.g. the value of a counter, the temperature measured by some sensor, or the latest input received by a reactive system. Trace variables $\pi, \tau, \pi', \tau', \ldots \in \mathcal{V}_t$ denote traces obtained by observing independent runs of some system or concurrent runs of individual subsystems. The syntax of Relational HyperLTL for a signature σ is:

$$\varphi ::= \forall \pi . \varphi \;\big|\; \exists \pi . \varphi \;\big|\; \psi \qquad \text{(formula)}$$

$$\psi ::= r(e, \ldots, e) \;\big|\; \neg \psi \;\big|\; \psi \vee \psi \;\big|\; \bigcirc \psi \;\big|\; \psi \,\mathsf{U}\, \psi \qquad \text{for } r \in S \qquad \text{(temporal formula)}$$

$$e ::= x_\pi \qquad \text{(expression)}$$

To interpret HyperLTL formulas over a structure $\mathcal{A}$, we use the alphabet $\Sigma = A^{\mathcal{V}_o}$. That is, events are valuations of the object variables in the domain of $\mathcal{A}$. A trace assignment $\Pi : \mathcal{V}_t \rightarrow \Sigma^\omega$ is a partial function mapping trace variables to infinite traces. We use $\Pi_\varnothing$ to denote the empty assignment, and $\Pi[\pi \mapsto t]$ for the same function as Π, except that π is mapped to trace t. The semantics of HyperLTL (over $\mathcal{A}$) is defined by associating formulas with pairs (T, Π), where T is a set of traces and Π is a trace assignment:

$$\begin{array}{lll} T, \Pi \models \forall \pi . \varphi & \text{iff} & \text{for all } t \in T, \text{ we have } T, \Pi[\pi \mapsto t] \models \varphi \\ T, \Pi \models \exists \pi . \varphi & \text{iff} & \text{there exists } t \in T, \text{ such that } T, \Pi[\pi \mapsto t] \models \varphi \\ T, \Pi \models \psi & \text{iff} & \Pi \models \psi \end{array}$$

The semantics of the temporal inner formulas is defined in terms of the traces associated with each path (here $\Pi[i, ..]$ denotes the map that assigns π to $t[i, ..]$ if $\Pi(\pi) = t$):

$$\begin{array}{lll}
\Pi \models r(e_1, \ldots, e_n) & \text{iff} & I_{\mathcal{A}}(r)([\![e_1]\!]_\Pi, \ldots, [\![e_n]\!]_\Pi) \text{ for } n = \mathrm{ar}(r) \\
\Pi \models \psi_1 \vee \psi_2 & \text{iff} & \Pi \models \psi_1 \text{ or } \Pi \models \psi_2 \\
\Pi \models \neg\psi & \text{iff} & \Pi \not\models \psi \\
\Pi \models \bigcirc\psi & \text{iff} & \Pi[1..] \models \psi \\
\Pi \models \psi_1 \,\mathsf{U}\, \psi_2 & \text{iff} & \text{for some } i, \Pi[i, ..] \models \psi_2, \text{ and} \\
 & & \text{for all } j < iT, \Pi[j, ..] \models \psi_1
\end{array}$$

The semantics of an expression e is simply the value associated with the corresponding pair of trace and object variables:

$$[\![x_\pi]\!]_\Pi = \Pi(\pi)[0](x)$$

We say that a set T of traces satisfies a HyperLTL formula φ (denoted $T \models \varphi$) if and only if $T, \Pi_\varnothing \models \varphi$. In the rest of the paper, for brevity, we refer to 'Relational HyperLTL' as just 'HyperLTL'.

The propositional semantics introduced for HyperLTL (see [19]) is a special case of the definition above, choosing $\mathcal{V}_o = \mathsf{AP}$ and the *Boolean model* $\mathcal{A} = 2$, which is defined as

$$\sigma_2 = (\{\mathsf{T}\}, \mathrm{ar}(\mathsf{T}) = 1) \qquad A_2 = \{\bot, \top\} \qquad I_2(\mathsf{T})(a) \text{ iff } a = \top.$$

Since there is only one relation symbol in this structure, we may as well drop it and simply write a_π instead of $\mathsf{T}(a_\pi)$, thus recovering the usual syntax of (propositional) HyperLTL. Note that in this domain all atomic relations can be constructed using Boolean connectives. For example, "π and π' agree on the value of a" can be written $(a_\pi \leftrightarrow a_{\pi'})$. In the remainder of the paper, we refer to HyperLTL over a structure $\mathcal{A}$ as HyperLTL$_{\mathcal{A}}$. If no particular structure is specified, we assume the usual syntax and semantics of HyperLTL$_2$.

Example 1 Recall the HyperLTL formulas (2) and (3) from the introduction

$$(2)\ \varphi_2 = \forall\pi.\forall\pi'.\square(☕_\pi \rightarrow ☕_{\pi'}) \qquad (3)\ \varphi_3 = \forall\pi.\exists\pi'.\square(\neg☕_\pi \rightarrow ☕_{\pi'})$$

and let $T = \{t_1, t_2, t_3\}$, where

$$t_1 = \{☕\}\{☕, 🍩\}\{🍩\}\cdots \qquad t_2 = \{☕, 🍩\}\{☕\}\{\}\cdots \qquad t_3 = \{☕\}\{☕\}\{☕\}\cdots.$$

and where the ellipses '$\cdots$' indicate that the last event is repeated indefinitely. Although traces t_1 and t_2 together satisfy (2), t_3 does not agree with the other two, i.e. $☕ \in t_3[2]$ but $☕ \notin t_1[2]$ and $☕ \notin t_2[2]$. Hence, $T \not\models \varphi_2$. On the other hand, $T \models \varphi_3$ because t_3 ensures that there is always coffee somewhere.

2.2 Semantic monitorability

Runtime verification is concerned with (1) generating a monitor from a formal specification φ, and (2) using the monitor to detect whether or not φ holds by observing events generated by the system at run time. Informally, monitorability refers to the feasibility of monitoring a property. Some properties are non-monitorable because no finite observation can lead to a conclusive verdict. We now present some abstract definitions to encompass previous notions of monitorability in a general way. These definitions are made concrete by instantiating them, for example, to traces (for trace properties) or sets of traces (for hyperproperties), see Ex. 2 below.

- **Observation.** We refer to the finite information provided dynamically to the monitor up to a given instant as an *observation*. We use O and O' to denote individual observations and $\mathcal{O}$ to denote the set of all possible observations. The set $\mathcal{O}$ is equipped with an *extension* order $O \preceq O'$.
- **System behavior.** We use $\mathcal{B}$ to denote the universe of all possible *behaviors* of a system. A behavior $B \in \mathcal{B}$ may, in general, be an infinite piece of information whereas an observation is always finite. By abuse of notation, $O \preceq B$ denotes that observation $O \in \mathcal{O}$ can be extended to a behavior B.
- **Property.** In this abstract sense, a *property* P is a predicate on $\mathcal{B}$: some behaviors satisfy the property P, while the rest violate it. We write $P(B)$ if the behavior B satisfies property P, and $\overline{P}(B)$ otherwise. We will use logics to specify properties of behaviors.

Example 2 When monitoring trace properties such as LTL, the set of observations is $\mathcal{O} = \Sigma^*$ because an observation is a finite trace $O \in \Sigma^*$. The order $\preceq$ is the prefix relation on strings. The set of behaviors is $\mathcal{B} = \Sigma^\omega$. Every formula φ induces a property P_φ on traces via its semantics: $P_\varphi(t)$ iff $t \models \varphi$.

When monitoring hyperproperties expressed in a logic such as HyperLTL, an observation is a finite set of finite traces $O \subset \Sigma^*$, that is, $\mathcal{O} = \mathcal{P}_{fin}(\Sigma^*)$. The order $\preceq$ is the prefix relation for finite sets of finite traces defined in the beginning of this section. That is, $O \preceq O'$ whenever for all $t \in O$ there is a $t' \in O'$ such that $t \preceq t'$. Behaviors are now sets of infinite traces, so $\mathcal{B} = \mathcal{P}(\Sigma^\omega)$. Finally, the semantics $\cdot \models \varphi$ of a HyperLTL formula φ defines a property on sets of traces.

We adapt the notions of satisfaction and violation of a property P from behaviors to observations as follows. An observation $O \in \mathcal{O}$ *permanently satisfies* a property P if every behavior $B \in \mathcal{B}$ that extends O satisfies P.

$$P(O) \quad \text{iff} \quad \text{for all } B \in \mathcal{B} \text{ such that } O \preceq B \text{ we have } P(B).$$

Similarly, an observation O permanently violates a property P if every extension $B \in \mathcal{B}$ of O violates P:

$$\overline{P}(O) \quad \text{iff} \quad \text{for all } B \in \mathcal{B} \text{ such that } O \preceq B \text{ we have } \overline{P}(B).$$

This slight abuse of notation is justified by the fact that an observation permanently violates a property if and only if it permanently satisfies the complement property (and vice versa). Obviously, no observation can permanently violate and permanently satisfy a given property. However, many observations neither permanently satisfy nor violate a given property, as different extensions of the observation have different outcomes with respect to the property.

These definitions can be instantiated to logics like LTL and HyperLTL. Given an LTL formula φ, a finite prefix u permanently satisfies φ if $v \models \varphi$ for all finite extensions $v \succeq u$, and u permanently violates φ if $v \not\models \varphi$ for all finite extensions $v \succeq u$. We write $u \models^s \varphi$ if u permanently satisfies φ, and $u \models^v \varphi$ if u permanently violates φ. The definitions and notations for permanent satisfaction and violation of HyperLTL formulas are obtained by replacing finite traces u, v with finite sets of finite traces U, V.

To monitor a system for satisfaction (or violation) of a formula φ is to decide whether a finite observation permanently satisfies (resp. violates) φ.

Definition 2 (semantic black-box monitorability) A property P is *(semantically) black-box monitorable* if, for every observation O, there exists an extended observation $O' \succeq O$, such that $P(O')$ or $\overline{P}(O')$.

Note that Def. 2 states that for every observation O there is *hope* in finding a final verdict by extending the observation. Instantiating Def. 2 for LTL with finite traces as observations ($\mathcal{O} = \Sigma^*$ and $\mathcal{B} = \Sigma^\omega$) leads to the traditional definition of monitorability for LTL by Pnueli and Zaks [38] (see also [13,26]) that states that a property is monitorable after observing O if O is not an ugly prefix, where a prefix is ugly if it cannot extended into a definite verdict. The fact that O is not an ugly prefix is called $\mathrm{PZ}(O)$ (for Pnueli-Zaks) in [2]. Note that a prefix is hopeless precisely when it is ugly. See Sect. 6 for a longer discussion on the different notions of monitorability, particularly the taxonomy discussed in [2].

Similarly, instantiating Def. 2 for HyperLTL with observations as finite sets of finite traces leads to monitorability as introduced by Agrawal and Bonakdarpour [3].

Example 3 The LTL formula $\square\diamond a$ is not (semantically) black-box monitorable since it requires an infinite-length observation, while formulas $\square a$ and $\diamond a$ are monitorable: $\square a$ is monitorable for violations as the monitor will flag a violation as soon as a is not seen, while an (acceptance) monitor for $\diamond a$ will signal acceptance when observing an a. Similarly, $\forall\pi.\forall\tau.\square(☕_\pi \rightarrow ☕_\tau)$ is (semantically) black-box monitorable for violation (it suffices to identify a case whenever there is coffee in the π trace but not in the corresponding place in the τ trace). On the other hand, $\forall\pi.\exists\tau.\square(\neg☕_\pi \rightarrow ☕_\tau)$ is not (semantically) black-box monitorable (neither for acceptance nor for violation) due to the quantifier alternation. We will prove this claim in detail in Sect. 3.

3 Improved monitorability by gray-box monitoring

Most of the previous definitions of monitorability in the literature assume one or a combination of the following:

- the logics are limited to trace logics, not considering hyperproperties [13,26,38];
- the system under analysis is black-box in the sense that the monitoring process must consider every further observation as a plausible extension [3,13,26], that is, the monitor makes decisions only based on the observed traces and it does not take into account the set of possible traces that the system can exhibit;
- the logics are tractable, in the sense that the decision problems of satisfiability, liveness, etc. are decidable for these logics. For example, one can decide the existence of an extended observation that satisfies or falsifies the given property [3,13,26,38].

We present here a more general notion of monitorability by challenging these assumptions. First, in Sect. 3.1 we prove negative results about the limitations of monitoring hyperproperties in a black-box manner. Then, in Sect. 3.2 we introduce the refined notionx of gray-box monitors and study the relation between their monitoring power with respect to semantic monitorability.

3.1 The limitations of (black-box) monitoring hyperproperties

Earlier work on monitoring hyperproperties is restricted to the quantifier alternation-free fragment, that is, considering only $\forall^*.\psi$ or $\exists^*.\psi$ properties. We establish now an impossibility result about the monitorability of formulas of the form $\forall\pi.\exists\pi'.\square\beta$, where β is a state predicate. For simplicity, we present the result for two traces, but this can be easily extended to arbitrarily many traces (two or more) and $\forall^+\exists^+$ hyperproperties. Intuitively speaking, the impossibility result works by first fixing a class of formulas, and then extending *any* observa-

tion into an extended observation that is satisfying (typically the set of all traces) and into a violating observation (by extending into a behavaior that fails to have a corresponding trace for the inner $\exists$ quantifier). Therefore, it is hopeless to monitor such formulas as one is unable to produce a definite verdict no matter what the observation is. Note that new observations both make it easier to have corresponding observations for the inner $\exists$, but create a further challenge for the outter $\forall$.

For a pair of traces π and π', a state predicate β is a relation over x_π and $x_{\pi'}$ or a Boolean combination of such relations, whereas it contains no temporal operators. Given a pair of valuations $v, v' \in \Sigma$, we write $\beta(v, v')$ as a shorthand for $\{\pi \mapsto vvv\cdots, \pi' \mapsto v'v'v'\cdots\} \models \beta$, i.e. β evaluated at v and v'. For example, the predicate $\beta = (a_\pi \leftrightarrow b_{\pi'})$ for $\mathcal{V}_o = \mathsf{AP} = \{a, b\}$ depends on the valuation of a and b at the first state of paths π and π', respectively.

A predicate β is *reflexive* if $\beta(v, v)$ holds for all valuations $v \in \Sigma$. A predicate β is *serial* if, for all v, there is a v' such that $\beta(v, v')$ holds. For example, the predicate $\beta \equiv \neg ☕_\pi \rightarrow ☕_{\pi'} \equiv ☕_\pi \vee ☕_{\pi'}$ is non-reflexive ($\beta(\emptyset, \emptyset)$ is false) and serial ($\beta(v, \{☕\})$ holds for any v). The following theorem captures precisely the monitorability of a safety $\forall\exists$ hyperproperty.

Theorem 1 *A HyperLTL$_\mathcal{A}$ formula of the form $\varphi = \forall\pi.\exists\pi'.\Box\beta$ is (semantically) black-box monitorable if and only if β is reflexive or non-serial.*

Proof Let $\varphi = \forall\pi.\exists\pi'.\Box\beta$. We show the two directions separately.

($\Rightarrow$) Assume φ is black-box monitorable. Then there must be a finite set V of finite traces (an extension of the empty set) such that $V \models^s \varphi$ or $V \models^v \varphi$. We analyze the two cases in turn.

- *Case* $V \models^s \varphi$. Pick an arbitrary $w \in \Sigma$ and let $u = www\cdots$ be the trace repeating w indefinitely. Extend V into the set $T = \{vu \mid v \in V\}$ of infinite traces. Let i be the length of the longest prefix in V. Then $t[i] = w$ for all $t \in T$. Since $V \models^s \varphi$, we have $T \models \varphi$ and, in particular, $t[i], t[i] \models \beta$. Hence $\beta(w, w)$ for any w, so β is reflexive.
- *Case* $V \models^v \varphi$. Because Σ^ω extends V, we must have $\Sigma^\omega \not\models \varphi$. Now assume that β is serial. Then the universal set Σ^ω is a model of φ because we can construct, for every trace $s \in \Sigma^\omega$, a trace $t \in \Sigma^\omega$ such that $\beta(s[i], t[i])$ for any i. Hence β cannot be serial.

($\Leftarrow$) Either φ is reflexive or non-serial. We consider each case in turn.

- If β is reflexive then φ holds for every non-empty set of infinite words by picking the same trace to instantiate π and π'. Therefore φ is black-box monitorable (in fact, guaranteed to be permanently satisfied for any observation).
- Assume that φ is non-serial. Then there must be a valuation $v \in \Sigma$ such that $\neg\beta(v, v')$ for any $v' \in \Sigma$. Consider an arbitrary observation U and extend one $u \in U$ into uv. The observation uv permanently violates φ because there is no candidate for π' that matches uv at the position where v occurs.

This concludes the proof. $\square$

In the proof of Theorem 1, the existence of an observation that permanently violates or permanently satisfies φ is sufficient to show reflexivity (for satisfaction) and non-seriality (for violation) of the state predicate. Similarly, in the other direction non-seriality implies black-box monitorability because for every observation there is an extended observation that

is a permanent violation. However, reflexivity implies a stronger notion of monitorability than that defined in Def. 2 as *every observation* is a permanent satisfaction. In other words, a future observation with a definite verdict is not merely possible, it is *guaranteed*.

The fragment of $\forall\exists$ properties captured by Theorem 1 is very general. First, the temporal fragment considered in Theorem 1 is safety, which are the simplest properties in terms of the temporal operators considered. Second, every binary predicate can be turned into a non-reflexive predicate by distinguishing the traces being related. In fact, many relational properties, such as non-interference and DDM, contain a tacit assumption that only distinct traces are being related and therefore the relation is non-reflexive. Seriality simply establishes that β cannot be falsified by only observing the local valuation of one of the traces. Intuitively, a non-serial predicate can be falsified by looking only at one of the traces, so the property is not a proper hyperproperty as it is not truly relational.

The proof of Theorem 1 makes essential use of the $\Box$ operator in the property $\forall\pi.\exists\pi'.\Box\beta$. One may be tempted to conclude that non-monitorability is therefore mainly a problem for temporal formulas. However, this is not the case. The following theorem shows that black-box monitorability is also impossible for a large class of completely *non-temporal* formulas.

We say that a subset $V \subseteq \Sigma$ of valuations is a *sink* of a state predicate β if, for all valuations $u \in \Sigma$, either $\beta(u, u)$ holds or there is a $v \in V$ such that $\beta(u, v)$ holds. If, in addition, V is finite, we say V is a *finite sink*.

Theorem 2 *A HyperLTL$_{\mathcal{A}}$ formula of the form $\varphi = \forall\pi.\exists\pi'.\beta$ is (semantically) black-box monitorable if and only if β is non-serial or has a finite sink.*

Proof Let $\varphi = \forall\pi.\exists\pi'.\beta$. We show the two directions separately.

($\Rightarrow$) Assume φ is black-box monitorable. Then, there must be a finite V (extending the empty set) such that $V \models^s \varphi$ or $V \models^v \varphi$. We analyze the two cases.

- *Case* $V \models^s \varphi$. Let $V[0] = \{v[0] \mid v \in V\}$. We show that $V[0]$ is a finite sink of β. Let $T = \{vvv\cdots \mid v \in V\}$. Pick an arbitrary $u \in \Sigma$, let $t = uuu\cdots$, and $T' = T \cup \{t\}$. Because V permanently satisfies φ and $V \preceq T'$, we have $T' \models \varphi$. In particular, $\{\pi \mapsto t, \pi' \mapsto t'\} \models \beta$ for $t' = t$ or $t' \in T$. If $t' = t$ then $\beta(u, u)$, otherwise $\beta(u, v[0])$ for some $v \in V$. Hence $V[0]$ is a finite sink of β.
- *Case* $V \models^v \varphi$. Because Σ^ω extends V, we must have $\Sigma^\omega \not\models \varphi$. Now assume that β is serial. Then the universal set Σ^ω is a model of φ because we can construct, for every trace $s \in \Sigma^\omega$, a trace $t \in \Sigma^\omega$ such that $\beta(s[0], t[0])$. Hence β cannot be serial.

($\Leftarrow$) Either φ is non-serial or φ has a finite sink. We consider each case.

- Assume φ is non-serial. Then there must be a valuation $u \in \Sigma$ such that $\neg\beta(u, v)$ for any $v \in \Sigma$. Hence, given any finite set U of finite traces, $U \cup \{u\}$ permanently violates φ.
- Assume φ has a finite sink W. Then W is a finite set of finite traces (each trace being a singleton) and W permanently satisfies φ since, for any trace t, there is a $w \in W \cup \{t[0]\}$ such that $\beta(t[0], w)$ holds. Given any other finite set U of finite traces, let $V = U \cup W$. Then V permanently satisfies φ.

$\square$

The practical consequence of Theorems 1 and 2 is that many hyperproperties involving one quantifier alternation cannot be monitored. We also note that Theorems 1 and 2 establish the minimal case for monitorability of HyperLTL formulas. Indeed, generalizing the $\forall\exists$ fragment

to single-alternation fragments with more quantifiers ($\forall^+\exists^+$) does not change proofs of Theorems 1 and 2.

Note that neither Theorem 1 nor Theorem 2 makes any assumptions on the choice of the structure $\mathcal{A}$ over which β is defined. Hence, the theorems hold both for propositional HyperLTL as well as any extensions to richer core logics. However, for finite domains A (and finite sets of object variables $\mathcal{V}_o$), Theorem 2 is trivial because the alphabet $\Sigma = A^{\mathcal{V}_o}$ is a finite sink of any serial state predicate β. This is true, in particular, for the propositional semantics of HyperLTL where $\Sigma = 2^{\mathsf{AP}}$. For richer core logics, on the other hand, Theorem 2 is non-trivial. We will see an example of this in Sect. 4, where we analyze monitorability of DDM.

Finally, note that even though the proofs of Theorem 1 and Theorem 2 have a very similar structure, neither of them subsumes the other.

3.2 Gray-box monitoring and the notions of sound and perfect monitors

We want to expand the spectrum of properties we could eventually monitor so we propose to exploit knowledge about the set of traces that the system can produce (gray-box or white-box monitoring). Given a system that can produce the set of system behaviors $\mathcal{S} \subseteq \mathcal{B}$, we parametrize the notions of permanent satisfaction and permanent violation to consider only behaviors in $\mathcal{S}$:

$$P_{\mathcal{S}}(O) \quad \text{iff} \quad \text{for all } B \in \mathcal{S} \text{ such that } O \preceq B \text{ we have } P(B),$$
$$\overline{P}_{\mathcal{S}}(O) \quad \text{iff} \quad \text{for all } B \in \mathcal{S} \text{ such that } O \preceq B \text{ we have } \overline{P}(B).$$

First, we extend the definition of monitorability (Def. 2 above) to consider the system under observation.

Definition 3 (semantic gray-box monitorability) A property P is *semantically gray-box monitorable* for a system $\mathcal{S}$ if every observation O has an extended observation $O' \succeq O$ in $\mathcal{S}$, such that $P_{\mathcal{S}}(O')$ or $\overline{P}_{\mathcal{S}}(O')$.

Example 4 Recall the hyperproperty $\varphi_3 = \forall\pi.\exists\tau.\Box(\neg\text{☕}_\pi \rightarrow \text{☕}_\tau)$ from our introductory example. We have already established that this property is not semantically black-box monitorable because the state predicate $\neg\text{☕}_\pi \rightarrow \text{☕}_\tau$ is non-reflexive and serial. A closer look at the proof of Theorem 1 reveals the exact culprit: φ_3 is not monitorable for violation because we cannot rule out the possibility that there will be a set extension in the future that adds the trace $t = \{\text{☕}\}\{\text{☕}\}\{\text{☕}\}\cdots$ which ensures that there is always coffee somewhere. This set extension, though permitted in theory, seems dubious in practice. The number of coffee dispensers is generally finite, even at the largest of conferences. If we constrain system behaviors $T \in \mathcal{S}$ to contain at most n traces, i.e. $|T| \leq n$ for all $T \in \mathcal{S}$, the property φ_3 becomes semantically gray-box monitorable for violation. To see this, let U be an arbitrary finite observation (containing at most n traces). To show that φ_3 is gray-box monitorable for violation, we need to exhibit a finite extension V of U that permanently violates φ_3. Let $U' \subset \Sigma^*$ be a set extension of U of size $|U'| = n$. Clearly, such an extension always exists since $\Sigma^* = \{\{\text{☕}\}, \varnothing\}^*$ contains an unbounded supply of distinct finite traces that can be added to U'. Denote the length of the longest trace in U' by l and let V be the set of finite traces obtained by padding every trace in U with $\varnothing$-events until the resulting trace v has length $|v| = l + 1$. Then $V \succeq U$ and V is a set of n traces, all of length $l + 1$, such that $\text{☕} \notin v[l]$ for any $v \in V$. Any extension T of V in $\mathcal{S}$ is necessarily a trace extension (as

opposed to a set extension). Hence, ☕ $\notin t[l]$ for any trace t in any such T, and therefore $V \models^{v}_{\mathcal{S}} \varphi_3$.

Gray-box monitorability strictly subsumes black-box monitorability as the latter corresponds to the special case where $\mathcal{S} = \mathcal{B}$. Hence, we will omit the qualifier "gray-box" when there is no risk of confusion. In cases where $\mathcal{S} \subset \mathcal{B}$, black-box monitorability is a (strictly) stronger property. This is a corollary of the following lemma.

Lemma 1 *Given a pair of systems $\mathcal{S}$, $\mathcal{S}'$ such that $\mathcal{S} \subseteq \mathcal{S}'$, any property P that is semantically monitorable for $\mathcal{S}'$ is also semantically monitorable for $\mathcal{S}$.*

Proof Let P be a property that is semantically monitorable for $\mathcal{S}'$. To show that P is also semantically monitorable for $\mathcal{S}$, let $O \in \mathcal{O}$ be an arbitrary observation. By assumption, O must have a finite extension $O' \in \mathcal{O}$ that either permanently satisfies of violates P for $\mathcal{S}'$. Assume $P_{\mathcal{S}'}(O')$ (the case for $\overline{P}_{\mathcal{S}'}(O')$ is analogous). Because any $B \in \mathcal{S}$ is also in $\mathcal{S}'$, we have $P(B)$ for any $B \succeq O'$ in $\mathcal{S}$, and hence $P_{\mathcal{S}}(O')$. □

Lemma 1 says that semantically monitorable properties P of a system S' remain monitorable for subsystems $S \subseteq S'$, but the converse does not hold in general: P need not be semantically monitorable for super-systems $S'' \supseteq S'$. Ex. 4 illustrates this: φ_3 is semantically monitorable for $S = \{T \mid |T| < n\}$ but not for $\Sigma^{\omega} \supseteq S$.

Following Def. 3, monitors must now analyze and decide properties of extended observations of a particular system. This, in turn, may not be computationally tractable for sufficiently rich system descriptions. To study this issue we introduce now a notion of monitors that consider $\mathcal{S}$ and the computational power of monitors (the diagonal dimension in Fig. 1). A *monitor* for a property P and a set of traces $\mathcal{S}$ is a *computable* function $M_{\mathcal{S}} : \mathcal{O} \to \{\top, \bot, ?\}$ that, given a finite observation O, decides a *verdict* for P:

- $\top$ indicates success,
- $\bot$ indicates failure, and
- ? indicates that the monitor cannot declare a definite verdict given only O.

To avoid clutter, we write M instead of $M_{\mathcal{S}}$ when the system is clear from the context. The following definition captures when a monitor for a property P can produce a definite answer.

Definition 4 (sound monitor) Given a property P and a set of behaviors $\mathcal{S}$, a monitor M is *sound* if, for every observation $O \in \mathcal{O}$,

1. if $M(O) = \top$, then $P_{\mathcal{S}}(O)$,
2. if $M(O) = \bot$, then $\overline{P}_{\mathcal{S}}(O)$.

If a monitor is not sound then it is possible that an extension of O forces M to change a $\top$ to a $\bot$ verdict, or vice-versa. The function that always outputs '?' is a sound monitor for any property, but this is the least informative monitor. A *perfect monitor* precisely outputs whether satisfaction or violation is inevitable, which is the most informative monitor.

Definition 5 (perfect monitor) Given a property P and a set of traces $\mathcal{S}$, a monitor M is *perfect* if, for every observation $O \in \mathcal{O}$,

1. if $P_{\mathcal{S}}(O)$, then $M(O) = \top$,
2. if $\overline{P}_{\mathcal{S}}(O)$, then $M(O) = \bot$,
3. otherwise $M(O) = ?$.

Obviously, a perfect monitor is sound. Similar definitions of perfect monitor only for satisfaction (resp. violation) can be given by forcing the precise outcome only for satisfaction (resp. violation).

Example 5 The following function is a monitor for φ_3 from Ex. 4:

$$M(U) = \begin{cases} ? & \text{if } |U| < n, \\ ? & \text{if, for all } i < \max\{|u| \mid u \in U\}, \text{ there is a } u \in U, \text{ s.t. } ☕ \in u[i], \\ \bot & \text{otherwise.} \end{cases}$$

where n is the maximum number of traces allowed by $\mathcal{S}$, i.e. $n = \max\{|T| \mid T \in \mathcal{S}\}$.

Clearly, M is computable. Furthermore, M is a sound monitor for φ_3, that is, $U \not\models^{v}_{\mathcal{S}} \varphi_3$ whenever $M(U) = \bot$. For M to produce a $\bot$ verdict, we must have $|U| = n$ and there must be an i such that ☕ $\notin u[i]$ for all $u \in U$. This means that any extension T of U in $\mathcal{S}$ is necessarily a trace extension, and therefore ☕ $\notin t[i]$ for every trace $t \in T$. Hence, U permanently violates φ_3.

If $n < \infty$, then M is also a perfect monitor for φ_3. First, note that no finite observation U can permanently satisfy φ_3 because we can extend every such U into a set of infinite traces that violates φ_3 by appending an infinite sequence of "¬☕" events to each trace in U. It remains to show that $U \not\models^{v} [\mathcal{S}]\varphi_3$ whenever $M(U) = ?$. There are two cases: if $|U| < n$, then we can always extend U into a set of traces $T \in \mathcal{S}$ that contains the trace $t = \{☕\}\{☕\}\{☕\}\cdots$ guaranteeing an indefinite supply of coffee; if, on the other hand, $|U| = n$ but there has always been coffee somewhere in U, i.e. for all $i < \max\{|u| \mid u \in U\}$, there is some $u \in U$ such that ☕ $\in u[i]$, then we can extend U into a set of infinite traces $T = \{ut \mid u \in U\}$, where again $t = \{☕\}\{☕\}\{☕\}\cdots$. In either case, $T \in \mathcal{S}$ and $T \models \varphi_3$.

A *black-box* monitor assumes that every behavior is potentially possible, that is $\mathcal{S} = \mathcal{B}$. If the monitor uses approximate information about the actual system, then we say it is *gray-box*. We use *white-box* when the monitor can reason with absolute precision about the set of traces of the system. In some cases, for example to decide instantiations of a $\forall$ quantifier, a satisfaction verdict that is taken from $\mathcal{S}$ can be concluded for all over-approximations (dually under-approximations for violation and for $\exists$).

Using Defs. 4 and 5, we now add the computability aspect to capture a stronger definition of monitorability. Abusing notation, we use $O \in \mathcal{S}$ to say that the observation O can be extended to a trace allowed by the system.

Definition 6 (strong monitorability) A property P is *strongly monitorable* for a system $\mathcal{S}$ if there is a sound monitor M such that for all observations $O \in \mathcal{O}$, there is an extended observation $O' \in \mathcal{S}$ for which either $M(O') = \top$ or $M(O') = \bot$.

It is easy to see that if a property is strongly monitorable (for $\mathcal{S}$) then it must be semantically monitorable. But the converse does not hold: in rich domains, some semantically gray-box monitorable properties may not be strongly monitorable. One simple example is the observation of deterministic Turing machines. Consider the problem monitoring whether a given deterministic Turing machine with a given input writes 1 in the tape infinitely many times, or only a finite number of times. Since there is only one trace, at a given point in time, there is only one behavior (because the machine is deterministic). Therefore the property is semantically monitorable. However, this property is not strongly monitorable as no such monitor exists.

In what follows we will use the term *monitorability* to refer to strong monitorability when there is no risk of confusion. Also, we say that a property is strongly monitorable

for satisfaction if the extension O' with $M(O') = \top$ always exists (and analogously for violation).

Lemma 2 *If P is strongly monitorable for a system $\mathcal{S}$, then P is semantically (gray-box) monitorable for $\mathcal{S}$.*

A property may not be monitorable in a black-box manner, but monitorable in a gray-box manner. In the realm of monitoring of LTL properties, strong and semantic monitorability coincide for finite state systems (see [43]) both in the case of black-box and in the case of gray-box for finite state systems, because model-checking and the problem of deciding whether a state of a Büchi automaton is live are decidable.

Following the idea sketched in [15] we propose to use a combination of static analysis and runtime verification to monitor violations of $\forall^+\exists^+$ properties (or dually, satisfactions of $\exists^+\forall^+$).

The main idea is to collect candidates for the outer $\exists$ part dynamically (traces from the real observed execution) and then use static analysis at runtime to over-approximate the inner $\forall$ quantifiers. For the latter we consider traces coming not only from the monitored system but also from other sources: they could be generated from a formal specification of the system, or from a model obtained from the system via symbolic execution and predicate abstraction. The key insight of our approach is that the current real execution of the system accounts for the outermost quantifier, while static analysis/verification is applied to explore runs of the model accounting for the innermost quantifier.

4 Monitoring distributed data minimality

In this section, we describe how to monitor DDM, which can be expressed as a hyperproperty of the form $\forall^+\exists^+$. The negative non-monitorability result from Sect. 3.1 can be generalized to $\forall^+\exists^+$ hyperproperties. In the particular case of DDM, although we mainly deal with the input/output relation of functions and are not concerned with infinite temporal behavior, we still need to handle possibly infinite set extensions $\mathcal{S}$ for black-box monitoring. In the remainder of this section, we discuss the following, seemingly contradictory aspects of DDM:

(P1) DDM is not semantically *black-box* monitorable,
(P2) DDM is semantically *white-box* monitorable (for programs that are not DDM),
(P3) checking DDM statically is undecidable,
(P4) DDM is strongly gray-box monitorable for violation, and we give a *sound monitor*.

The apparent contradictions are resolved by careful analysis of DDM along the different dimensions of the monitorability cube (Fig. 1).

We will show how to monitor DDM and similar hyperproperties using a gray-box approach. In our approach, a monitor can decide the existence of traces at run time using a limited form of static analysis. The static analyzer receives the finite observation O collected by the monitor, but not the future system behavior. Instead it must reason under the assumption that any system behavior in $\mathcal{S}$ that is compatible with O may eventually occur. For example, given an $\exists\forall$ formula, the outer existential quantifier is instantiated with a concrete set U of runtime traces, while possible extensions of U provided by static analysis can be used to instantiate the inner universal quantifier.

```
1  class Toll {
2    int rate(int hour, int passengers) {
3      int r;                                      // standard rates:
4      if (hour >= 9 && hour <= 17) { r = 90; }    //  - daytime
5      else                         { r = 70; }    //  - nighttime
6      if (passengers > 2) { r = r - (r / 5); }    // carpool: 20return r;
7    }
8    int max(int x, int y) {
9      if (x > y) { return x; } else { return y; }
10   }
11   int fee(int t1, int t2, int t3, int p) {
12     int r1 = rate(t1, p);          // rates at each toll station
13     int r2 = rate(t2, p);
14     int r3 = rate(t3, p);
15     int f1 = max(r1, r2) * 4;      // fees per road section
16     int f2 = max(r2, r3) * 7;
17     return f1 + f2;                // total fee
18   }
19 }
20
```

Fig. 2 A program for computing the total fee of a trip on a toll road

4.1 DDM preliminaries

We briefly recapitulate the concepts of data minimization and data minimality [7]. Before giving their formal definitions, let us illustrate the ideas behind these concepts on the short Java program shown in Fig. 2. Consider the method `rate`. Its purpose is to compute the baseline rate to be paid by the driver of a vehicle on a toll road. The rate depends on the time of day and the number of passengers in the vehicle. The range of the output is {56, 70, 72, 90}, and consequently the data processor does not need to know the precise hour of the day, nor the exact number of passengers. A vehicle might pass a toll station at any time between 9pm and 5am to be subject to the higher daytime rates (72, 90), and at any other time to benefit from the lower nighttime rates (56, 70). Also, any vehicle occupied by three or more passengers is eligible for a 20% carpool discount. Giving the actual hour and number of passengers violates the principle of data minimality because more information than necessary is collected. Data minimization is the process of ensuring that the range of inputs provided is reduced, such that different inputs result in different outputs.

Formally, given a function $f : I \to O$, the problem of data minimization consists in finding a *preprocessor* function $p : I \to I$, such that $f = f \circ p$ and $p = p \circ p$. The goal of p is to limit the information available to f while preserving the behavior of f. There are many possible such preprocessors (e.g. the identity function), which can be ordered according to the information they disclose, that is, according to the subset relation on their *kernels*. The kernel $\ker(p)$ of a function p is defined as the equivalence relation $(x, y) \in \ker(p)$ iff $p(x) = p(y)$. The smaller $\ker(p)$ is, the more information p discloses. The identity function is the worst preprocessor since it discloses all information (its kernel is equality—the least equivalence relation). An optimal preprocessor, or *minimizer*, is one that discloses the least amount of information.

A function f is *monolithic data-minimal* (MDM), if it fulfills either of the following equivalent conditions:

1. the identity function is a minimizer for f,
2. f is injective.

Seeing as MDM is just injectivity (by Condition 2), one may wonder why we went through the effort of defining preprocessors and their information order only to arrive at an equivalent but more convoluted definition (Condition 1). The reason is that, besides MDM, there are other variants of data minimality (e.g. DDM), whose logical characterizations are not as intuitive. The advantage of Condition 1 is that it is an information-flow-based characterization that can be generalized to more complicated settings in a relatively straightforward fashion (as we will see shortly). Condition 2, on the other hand, is a purely logical or *data-based* characterization suitable for implementation in e.g. a monitor.

MDM is the strongest form of data minimality, where one assumes that all input data is provided by a single source and thus a single preprocessor can be used to minimize the function. In a distributed setting, the concept of minimization is more complex as input data may be collected from multiple independent sources. Consider the method `fee` in Fig. 2. This method computes the total fee for a trip on a toll road, based on the hours at which a vehicle passes three consecutive toll stations, and on the number of passengers in the vehicle. The overall fee depends on the total time spent on the toll road, which is data collected from all three toll stations. In particular, if a vehicle enters a section of the toll road during a low-rate early morning hour, but fails to reach the next station before 9pm, the driver will be charged the more expensive daytime rate for the entire section. DDM requires minimizing each input parameter (i.e. the information collected at separate toll stations) individually. A preprocessor located at any given toll station can easily minimize the individual inputs (`hour`, `passengers`) at that station. But an individual preprocessor cannot guarantee minimization with respect to the overall fee since it has no information about the input data collected at the other stations. DDM therefore constitutes merely a "best effort" to minimize inputs given the inherently distributed nature of the system.

As a second example, consider a web-based auction system that accepts bids from n bidders, represented by distinct input domains $I_1, \ldots, I_n$, and where concrete bids $x_k \in I_k$ are submitted remotely. The auction system must compute the function $m(x_1, \ldots, x_n) = \max_k\{x_k\}$, which is clearly non-injective and, hence, non-MDM. In this case, a single, monolithic minimizer cannot be used since different bidders need not have any knowledge of each other's bids. Instead, bidders must try to minimize the information contained in their bid locally, in a distributed way, before submitting it to the auction.

The problem of *distributed data minimization* consists in building a collection $p_1, \ldots, p_n$ of n independent preprocessors $p_k\colon I_k \to I_k$ for a given function $f\colon I_1 \times \cdots \times I_n \to O$, such that their product $p(x_1, \ldots, x_n) = (p_1(x_1), \ldots, p_n(x_n))$ is a preprocessor for f. Such preprocessors are called *distributed*, and a distributed preprocessor for f that discloses the least amount of information is called a *distributed minimizer* for f. Then, one can generalize the (information-flow) notion of data-minimality to the distributed setting as follows. The function f is *distributed data-minimal* (DDM) if the identity function is a distributed minimizer for f. For example, the maximum function m defined above is DDM. As for MDM, there is an equivalent, data-based characterization of DDM described next.

Proposition 1 *(distributed data minimality [7,35]) A function f is DDM iff, for all input positions k and all $x, y \in I_k$ such that $x \neq y$, there is some $z \in I$, such that $f(z[k \mapsto x]) \neq f(z[k \mapsto y])$.*

We refer the interested reader to Antignac et al. [7] for a more detailed discussion of data minimality and for a detailed proof of Proposition 1.

The alternative characterization of DDM given in Prop. 1 serves as our formal specification for exploring the monitorability of DDM. In the following, we assume that the function $f\colon I_1 \times \cdots \times I_n \to O$ has at least two arguments ($n \geq 2$). Note that for unary functions, DDM

coincides with MDM. Since MDM is a $\forall^+$-property (involving no quantifier alternations), most of the challenges to monitorability discussed here do not apply [36]. We also assume, without loss of generality, that the function f being monitored has only nontrivial input domains, i.e. $|I_k| \geq 2$ for all $k = 1, \ldots n$. If I_k is trivial then this constant input can be ignored. Finally, note that checking DDM statically is undecidable (P3) for sufficiently rich programming languages [7].

4.2 DDM as a hyperproperty

We consider data-minimality for total functions $f: I \to O$. The domain of interpretation A_f is the set of possible *input-output (I/O) pairs* of f, i.e. $A_f = I \times O$. We do not distinguish between different observables (each I/O pair is an observation in itself). Hence our alphabet, or set of events, is just $\Sigma_f = A_f = I \times O$, i.e. the set of object variables is the singleton set $\mathcal{V}_o = \{*\}$. Since a single I/O pair $u \in \Sigma_f$ captures an entire run of f, we restrict ourselves to observing singleton traces, i.e. traces of length $|u| = 1$. In other words, we ignore any temporal aspects associated with the computation of f. This allows us to use first-order predicate logic—without any temporal modalities—as our specification logic.

DDM is a hyperproperty, expressed as a predicate over sets of traces, even though the traces are I/O pairs. The set of observable behaviors $\mathcal{O}_f$ of a given f consists of all *finite sets* of I/O pairs $\mathcal{O}_f = \mathcal{P}_{fin}(\Sigma_f)$. The set of all possible system behaviors $\mathcal{B}_f = \mathcal{P}(\Sigma_f)$ additionally includes *infinite sets* of I/O pairs.

Example 6 Let $f: \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ be the addition function on natural numbers, $f(x, y) = x + y$. Then $I = \mathbb{N} \times \mathbb{N}$, $O = \mathbb{N}$, and a valid trace $u \in \Sigma_f$ takes the form $u = ((x, y), z)$, where x, y and z are all naturals. Both $U = \{((1, 2), 3), ((2, 1), 3)\}$ and $V = \{((1, 1), 3)\}$ are considered observable behaviors $U, V \in \mathcal{O}_f$, even though V does not correspond to a valid system behavior since $f(1, 1) \neq 3$. Remember that we do not discriminate between valid and invalid system behaviors in a black-box setting.

For a function f with n inputs, we define the set of binary relation symbols

$$S_f = \{\text{output}\} \cup \bigcup_{i=1}^{n} \{\text{same}_i, \text{almost}_i\}.$$

This induces a signature $\sigma_f = (S_f, \text{ar}(r) = 2)$. Given a tuple $x = (x_1, x_2, \ldots, x_n)$, we write $\text{proj}_i(x)$ or simply x_i for its i-th projection. Given an I/O pair $u = (x, y)$, we use $\text{in}(u)$ for the input component and $\text{out}(u)$ for the output component (i.e. $\text{in}(u) = x$ and $\text{out}(u) = y$). The interpretation of the relations in S_f is

$$\begin{aligned}
I(\text{output})(u, v) &\equiv \text{out}(u) = \text{out}(v) && u \text{ and } v \text{ agree on their output,} \\
I(\text{same}_i)(u, v) &\equiv \text{in}(u)_i = \text{in}(v)_i && u \text{ and } v \text{ agree on the } i\text{-th input,} \\
I(\text{almost}_i)(u, v) &\equiv \bigwedge_{k \neq i} \text{in}(u)_k = \text{in}(v)_k && u \text{ and } v \text{ agree on all but the} i\text{-th input}
\end{aligned}$$

Example 7 Let $\Pi = \{\pi \mapsto ((1, 2), 3), \pi' \mapsto ((2, 1), 3)\}$. Then $\Pi \models \text{output}(\pi, \pi')$, but $\Pi \not\models \text{same}_1(\pi, \pi')$ and $\Pi \not\models \text{almost}_1(\pi, \pi')$.

We define DDM for input argument i as follows:

$$\varphi_i = \forall \pi. \forall \pi'. \exists \tau. \exists \tau'. \neg\, \text{same}_i(\pi, \pi') \to \begin{pmatrix} \text{same}_i(\pi, \tau) \wedge \text{same}_i(\pi', \tau') \wedge \\ \text{almost}_i(\tau, \tau') \wedge \neg\, \text{output}(\tau, \tau') \end{pmatrix}$$

In words: given any pair of traces π and π', if the inputs of π and π' differ in their i-th position, then there must be some common values z for the remaining inputs, such that the outputs of f for $\text{in}(\tau) = z[i \mapsto \text{in}(\pi)_i]$ and $\text{in}(\tau') = z[i \mapsto \text{in}(\pi')_i]$ differ. Note that z does not appear in φ_i directly, instead it is determined implicitly by the (existentially quantified) traces τ and τ'. Finally, *distributed data minimality* for f is defined as

$$\varphi_{\mathsf{dm}} = \bigwedge_{i=1}^{n} \varphi_i .$$

The property φ_{dm} follows the same structure as the logical characterization of DDM given in Prop. 1. The universally quantified variables range over the possible inputs at position i, while the existentially quantified variables τ and τ' range over the other inputs and the outputs. Note also that, given the input coordinates of π, π', and τ, all the output coordinates, as well as the input coordinates of τ', are uniquely determined. Note that even though φ_{dm} is not in prenex normal form, it is a finite conjunction of $\forall\forall\exists\exists$ formulas in prenex normal form, so a finite number of monitors can be built and executed in parallel, one per input argument.

Example 8 Consider again $U = \{((1, 2), 3), ((2, 1), 3)\}$ and $V = \{((1, 1), 3)\}$ from Ex. 6. Then, $V \models \varphi_{\mathsf{dm}}$ trivially holds, but $U \not\models \varphi_{\mathsf{dm}}$ because when $\Pi(\pi) \neq \Pi(\pi')$ there is no choice of $\Pi(\tau), \Pi(\tau') \in U$ for which $\Pi \models \neg\, \text{output}(\tau, \tau')$ holds.

Note that, in the above example, $V \models \varphi_{\mathsf{dm}}$ holds despite the fact that V is not a valid behavior of the example function $f(x, y) = x + y$. Indeed, whether or not $U \models \varphi_{\mathsf{dm}}$ holds for a given U is independent of the choice of f. In particular, $\Sigma_f \models \varphi_{\mathsf{dm}}$, for any choice of f regardless of whether f is data-minimal or not. This is already a hint that the notion of semantic black-box monitorability is too weak to be useful when monitoring φ_{dm}. Since Σ_f is a model of φ_{dm}, no observation U can have an extension that permanently violates φ_{dm}. As we will see shortly, gray-box monitoring does not suffer from this limitation. Monitorability of DDM for violations becomes possible once we exclude potential models such as Σ_f which do not correspond to valid system behaviors.

Remark Note that although our definition and approach work for general (reactive) systems, the DDM example is admittedly a non-reactive system with traces of length 1. This, however, is not a limitation of the approach. Extending DDM for reactive systems is left as future work.

4.3 Properties of DDM

Since φ_{dm} is a $\forall^+\exists^+$ property, it should not come as a surprise that it is *not* semantically black-box monitorable in general (P1). Lemma 3 below essentially follows from Theorem 2.

Lemma 3 *(black-box non-monitorability) Assume $f : I \to O$, then φ_{dm} is semantically black-box monitorable iff I is finite.*

Proof Although Theorem 2 does not apply exactly to the current setting—φ_{dm} has four instead of two quantifiers—the proof goes through with only minor adjustments. We first establish that φ_{dm} is serial (in the appropriate sense), from which it follows that φ_{dm} has a finite sink if and only if Σ_f is finite. The remainder of the proof then follows the same argument as that of Theorem 2.

Without loss of generality, assume that f is surjective. (Otherwise, replace the codomain O with the image of f in the remainder of the proof.)

We first show that φ_{dm} is serial. Assume I and O each contain at least two elements. Smaller I/O domains correspond to degenerate cases for which semantic black-box monitorability is easy to show, so we omit them here. Let u, u' be arbitrary I/O pairs, $o \neq o' \in O$ a pair of distinct outputs, and i an arbitrary input position. Define $v = (\text{in}(u), o)$ and $v' = (\text{in}(u)[i \mapsto \text{in}(u')_i], o')$. Then u, u' and v, v' are all in Σ_f, and it is easy to check that φ_{dm} holds if the quantified variables are instantiated to these traces in the given order. In other words, φ_{dm} is serial in the intuitive sense: for any instantiation of the universally quantified variables to valuations in Σ_f, there is at least one corresponding instantiation of the existentially quantified variables. (Note that this makes Σ_f a sink of φ_{dm} in a similar sense.)

If I is finite, then (by assumption) so is Σ_f, and hence φ_{dm} has a "finite sink". Concretely, Σ_f is a finite extension of any finite observation U, and it is the largest such observation, so it permanently satisfies φ_{dm}. It follows that φ_{dm} is semantically black-box monitorable for satisfaction.

Assume instead that I is infinite, and let U be any finite set of traces. To show that U neither permanently satisfies nor permanently violates φ_{dm}, it is sufficient to exhibit a pair of extensions $T_s, T_v \succeq U$ that satisfy and violate φ_{dm}, respectively. For T_s, we pick $T_s = \Sigma_f$. Since φ_{dm} is serial, $T_s \models^s \varphi_{\mathsf{dm}}$.

We have to work slightly harder to construct T_v. Intuitively, we must show that φ_{dm} does not have a "finite sink". Concretely, we show that any finite observation U can be extended with a pair of traces v, v' such that $U \cup \{v, v'\}$ violates φ_{dm}.

Since I is infinite but U is finite, there must be an input position i and a pair of distinct elements $x \neq x' \in I_i$ such that no trace in U has x or x' as its i-th input. Pick some arbitrary trace $w \in \Sigma_f$, and let $v = w[i \mapsto x]$ and $v' = w[i \mapsto x']$. By construction, $v, v' \notin U$, so $T_v = U \cup \{v, v'\}$ is a strict extension of U. To show that T_v does indeed violate φ_{dm}, it is sufficient to show that $T_v \models^v \varphi_i$. Pick v, v' to instantiate π and π'. Then $\text{in}(w)_i = x \neq x' = \text{in}(w')_i$ by construction, but there is no way to instantiate τ and τ': since they have to agree with π and π' on the i-th input position, the only candidates are v and v', but $\text{out}(v) = \text{out}(v')$ by construction. □

Perhaps surprisingly, φ_{dm} is semantically *white-box* monitorable for violations (P2). That is, if f is not DDM, there is hope to detect it. To make this statement more precise, we first need to identify the set of valid system behaviors $\mathcal{S}_f$ of f. We define $\Sigma_f^{\#} = \{(x, y) \mid f(x) = y\}$ to be the set of I/O pairs that correspond to executions of f. Then $\mathcal{S}_f = \mathcal{P}(\Sigma_f^{\#})$ precisely characterizes the set of valid system behaviors.

Example 9 Define $g \colon \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ as $g(x, y) = x$, i.e. g simply ignores its second argument. Then $\Sigma_g^{\#} = \{((x, y), x) \mid x, y \in \mathbb{N}\}$. It is easy to show that DDM is white-box monitorable for g. Any finite set of valid traces U can be extended to include a pair of traces u, u' that only differ in their second input value, e.g. $u = ((1, 1), 1)$ and $u' = ((1, 2), 1)$. Now, consider any $T \in \mathcal{S}_f$ that extends $U \cup \{u, u'\}$. Clearly, T cannot contain any trace v for which $\text{in}(v)_1 = 1$ but $\text{out}(v) \neq 1$ as that would constitute an invalid system behavior. But T would have to contain such a trace to be a model of φ_2. Hence, $T \not\models \varphi_{\mathsf{dm}}$ for any such T, which means $U \cup \{u, u'\}$ permanently violates φ_{dm}.

Note the crucial use of information about g in the above example: it is the restriction to *valid* extensions $T \in \mathcal{S}_f$ that excludes trivial models such as Σ_f and thereby restores (semantic) monitorability for violations. The apparent conflict between (P1) and (P2) is thus resolved.

With the extra information that gray-box monitoring affords, we can make more precise claims about properties like DDM: whether or not a property is monitorable may, for instance,

depend on whether the property actually holds for the system under scrutiny. Concretely, for the case of DDM, we show the following.

Theorem 3 *Given a function $f : I \to O$, the formula φ_{dm} is semantically gray-box monitorable in $\mathcal{S}_f$ if and only if either f is distributed non-minimal or the input domain I is finite.*

Theorem 3 follows from the following two auxiliary lemmas.

Lemma 4 (semantic violation) *If f is not DDM, then φ_{dm} is semantically monitorable for violation (in $\mathcal{S}_f$).*

Proof Assume a finite set of traces $U \in \mathcal{S}_f$. We need to show that there is a finite extension $V \succeq U$ permitted by $\mathcal{S}_f$ that permanently violates φ_{dm}. First, note that the task is trivial if I is finite: we simply pick $V = \Sigma^{\#}_f$, i.e. the set of all possible executions, which is also finite. The only finite extension of V permitted by $\mathcal{S}_f$ is the complete set of traces $\Sigma^{\#}_f$ itself, and since f is not distributed minimal, φ_{dm} cannot hold for $\Sigma^{\#}_f$.

Assume instead that I is infinite. Since f is distributed non-minimal, there must be some input position i and some pair of distinct inputs $x \neq x' \in I_i$, such that $f(z[i \mapsto x]) = f(z[i \mapsto x'])$ for any choice of $z \in I$. Let $y = z[i \mapsto x]$ and $y' = z[i \mapsto x']$ for an arbitrary $z \in I$. Then any set $W \in \mathcal{S}_f$ that contains the traces $u = (y, f(y))$ and $u' = (y', f(y'))$ violates φ_{dm}. To see this, assume instead that $W \models^s_{\mathcal{S}_f} \varphi_{\mathsf{dm}}$. Then there must be traces $v, v' \in W$ that agree on all but the i-th input, such that $f(\mathrm{in}(v)[i \mapsto x]) \neq f(\mathrm{in}(v')[i \mapsto x'])$, thus contradicting non-minimality of f. Hence, by picking $V = U \cup \{u, u'\}$, we have $V \models^v_f \varphi_{\mathsf{dm}}$. □

Lemma 5 (Semantic satisfaction) *If $f : I \to O$ is DDM, then φ_{dm} is semantic monitorable for satisfaction (in $\mathcal{S}_f$) if and only if I is finite.*

Proof First, if I is finite the result follows by picking $V = \Sigma^{\#}_f$. Assume now that f is distributed minimal, φ_{dm} is semantically monitorable for satisfaction, and I is infinite. Let $U \in \mathcal{S}_f$ be some non-empty, finite set of traces with some distinguished element $u \in U$. Since φ_{dm} is monitorable for satisfaction, there must be a finite extension $V \succeq U$ that permanently satisfies φ_{dm}. To arrive at a contradiction, it suffices to construct a finite extension $W \succeq V$ that does not satisfy φ_{dm}.

Pick an input position i for which I_i is infinite. Such an i must exist because otherwise I would be the Cartesian product of finite sets, and I is infinite by assumption. Next, pick a pair of distinct element $x \neq x' \in I_i$ such that there are no traces in V with x or x' as their i-th input. Such x, x' must also exist because I_i is infinite but V is finite. Finally, pick an input position $j \neq i$, and a $y \in I_j$ such that $y \neq \mathrm{in}(u)_j$. Such a y must exist for I_j to be non-trivial.

Now let $z = \mathrm{in}(u)[i \mapsto x]$, $z' = \mathrm{in}(u)[i \mapsto x', j \mapsto y]$ and $w = (z, f(z))$, $w' = (z', f(z'))$. Then w and w' are clearly valid traces, i.e. $w, w' \in \Sigma^{\#}_f$, but $w, w' \notin V$ since w and w' have x and x' as their i-th inputs, respectively. Let $W = V \cup \{w, w'\}$. By construction, $\neg\, \mathrm{same}_i(\pi, \pi')$ holds if we instantiate π and π' to w and w', respectively, but there is no pair of traces $v, v' \in W$ to instantiate τ, τ' in such a way that $\mathrm{same}_i(\pi, \tau)$, $\mathrm{same}_i(\pi', \tau')$ and $\mathrm{almost}_i(\tau, \tau')$ all hold simultaneously. The former force the choice $\tau \mapsto w$ and $\tau' \mapsto w'$ but, by construction, $\mathrm{in}(w)_j \neq \mathrm{in}(w')_j$. Hence $W \not\models^s \varphi_{\mathsf{dm}}$ and we arrive at a contradiction.
□

Intuitively, Theorem 3 means that f cannot be monitored for satisfaction. Note that the semantic monitorability property established by Theorem 3 is independent of whether we can actually decide DDM for the given f. We address the question of strong monitorability later on in this section.

If I is finite, it is easy to strengthen Theorem 3 by providing a perfect monitor M_{dm} for φ_{dm}. Since f is assumed to be a total function with a finite domain, we can simply check the validity of φ_{dm} for every trace $U \subseteq \Sigma_f^{\#}$ and tabulate the result. To do so, the $\exists$ and $\forall$ quantifiers in φ_{dm} can be converted into conjunctions and disjunctions over U.

Corollary 1 *For $f : I \to O$ with finite I, φ_{dm} is strongly monitorable in $\mathcal{S}_f$.*

If I is infinite, then φ_{dm} is not semantically monitorable for satisfaction, but we can still hope to build a sound monitor for violation of φ_{dm}.

4.4 Building a gray-box monitor for DDM

In what follows, we assume a computable function capable of deciding DDM only for some instances. This function, which we call oracle, will serve as the basis for a *sound* monitor for DDM (P4). This monitor will detect some, but not all, violations of DDM when given sets of observed traces, thus resolving the apparent tension between (P3) and (P4).

Given $f\colon I_1 \times \cdots \times I_n \to O$, we define the predicate φ_f as

$$\varphi_f(i, x, y) = \exists z \in I.\, f(z[i \mapsto x]) \neq f(z[i \mapsto y]), \tag{1}$$

and assume a total computable function $N_{f,i}\colon I_i \times I_i \to \{\top, \bot, ?\}$ such that

$$N_{f,i}(x, y) = \begin{cases} \top \text{ or } ? & \text{if } \varphi_f(i, x, y) \text{ holds,} \\ \bot \text{ or } ? & \text{otherwise.} \end{cases}$$

The function $N_{f,i}$ acts as our oracle to instantiate the existential quantifiers in φ_{dm}. As discussed earlier, such oracles may be implemented by statically analyzing the system under observation (here, the function f). In our proof-of-concept implementation, we extract φ_f from f using symbolic execution, and we use an SMT solver to compute $N_{f,i}$ (see Sect. 5 for details).

We now define a monitor M_{dm} for φ_{dm} as follows:

$$M_{\mathsf{dm}}(U) = \begin{cases} ? & \text{if } f(\mathrm{in}(u)) \neq \mathrm{out}(u) \text{ for some } u \in U, \\ ? & \text{if } \bigwedge_{i=1}^{n} \bigwedge_{u,u' \in U} N_{f,i}(\mathrm{in}(u)_i, \mathrm{in}(u')_i) \neq \bot, \\ \bot & \text{otherwise.} \end{cases}$$

Intuitively, the monitor $M_{\mathsf{dm}}(U)$ checks the set of traces U for violations of DDM by verifying two conditions: the first condition ensures the *consistency* of U, i.e. that every trace in U does in fact correspond to a valid execution of f; the second condition is necessary for U *not* to permanently violate φ_{dm}. Hence, if it fails, U must permanently violate φ_{dm}. Since $N_{f,i}$ is computable, so is M_{dm}. Note that M_{dm} never gives a positive verdict $\top$. This is a consequence of Theorem 3: if f is DDM, then φ_{dm} is not monitorable in $\mathcal{S}_f$. In other words, DDM is not monitorable for satisfaction.

The second condition in the definition of M_{dm} is an approximation of φ_{dm}: the universal quantifiers are replaced by conjunctions over the finite set of input traces U, while the existential quantifiers are replaced by a single quantifier ranging over all of $\Sigma_f^{\#}$ (not just U). This approximation is justified formally by the following theorem.

Theorem 4 *(soundness) The monitor M_{dm} is sound. Formally,*

1. $U \models^{s}_{\mathcal{S}_f} \varphi_{\mathsf{dm}}$ *if* $M_{\mathsf{dm}}(U) = \top$*, and*
2. $U \models^{v}_{\mathcal{S}_f} \varphi_{\mathsf{dm}}$ *if* $M_{\mathsf{dm}}(U) = \bot$.

Proof The monitor never gives a $\top$ verdict, so the first half of the theorem (satisfaction) holds vacuously. For the second part (violation), we have

$$M_{\mathsf{dm}}(U) = \bot \quad \Leftrightarrow \quad U \in \mathcal{S}_f \;\wedge\; \bigvee_{i=1}^{n} \bigvee_{u,u' \in U} N_{f,i}(\mathrm{in}(u)_i, \mathrm{in}(u')_i) = \bot,$$

and

$$\begin{aligned}
& \bigvee_{i=1}^{n} \bigvee_{u,u' \in U} N_{f,i}(\mathrm{in}(u)_i, \mathrm{in}(u')_i) = \bot \\
\Leftrightarrow\;& \bigvee_i \exists u, u' \in U.\, \neg\varphi_f(i, \mathrm{in}(u)_i, \mathrm{in}(u')_i) \\
\Leftrightarrow\;& \bigvee_i \exists u, u' \in U.\, \forall z \in I.\, f(z[i \mapsto \mathrm{in}(u)_i]) = f(z[i \mapsto \mathrm{in}(u')_i]) \\
\Leftrightarrow\;& \bigvee_i \exists u, u' \in U.\, \forall w \in \Sigma_f^{\#}.\, f(\mathrm{in}(w)[i \mapsto \mathrm{in}(u)_i]) = f(\mathrm{in}(w)[i \mapsto \mathrm{in}(u')_i]) \\
\Rightarrow\;& \forall V \in \mathcal{S}_f.\, U \preceq V \Rightarrow \\
& \quad \bigvee_i \exists u, u' \in V.\, \forall w \in V.\, f(\mathrm{in}(w)[i \mapsto \mathrm{in}(u)_i]) = f(\mathrm{in}(w)[i \mapsto \mathrm{in}(u')_i]) \\
\Leftrightarrow\;& U \models^{v}_{\mathcal{S}_f} \varphi_{\mathsf{dm}}.
\end{aligned}$$

□

In the next section, we describe a prototype implementation of M_{dm}.

5 Implementation and empirical evaluation

We have implemented the ideas described in Sect. 4 in a proof-of-concept monitor for DDM called *Minion*. The monitor uses the symbolic execution API and the SMT backend of the KeY deductive verification system [4,27] to extract logical characterizations of Java programs. It then extends them to first-order formulas over sets of observed traces, and checks the result using the state-of-the-art SMT solver Z3 [32,33]. *Minion* is written in Scala and provides a simple command-line interface. Its source code is freely available online at https://github.com/sstucki/minion/.

In the remainder of this section, we describe *Minion* in more detail and illustrate its behavior on concrete example programs. We start with a high-level summary of symbolic execution and how to use it for extracting logical characterizations of programs. We present the different monitoring strategies implemented in *Minion* and illustrate how our tool deals with complex control flow such as loops. We conclude the section by presenting a short empirical evaluation of *Minion*, where we tested the two monitoring strategies for DDM implemented in our tool on different workloads. Note that this evaluation does not constitute a performance benchmark as performance has not been the focus of our proof-of-concept implementation.

5.1 Extracting program characterizations via symbolic execution

Minion extracts logical characterization of programs via *symbolic execution*, a process that systematically explores the execution paths of a program on all possible inputs [16,28]. During symbolic execution, program inputs are kept abstract (symbolic) and the program

state (i.e. the values of local variables and memory locations) is represented by expressions over these abstract inputs. When the program branches on an abstract value, the symbolic execution engine explores both branches, keeping track of the *path conditions* that must hold in the respective branches. The output of symbolic execution is a so-called *symbolic execution tree*. Each branch of the tree corresponds to possible execution path p of the program, and each leaf summarize the final state s_p of the corresponding path and the path condition c_p under which it was obtained. The symbolic execution tree thus constitutes a finite representation of all possible program behaviors, from which we can extract a first-order formula $\psi = \bigvee_p c_p \wedge s_p$ describing the overall program behavior (cf. [7], Def. 8). To build a gray-box monitor for DDM as described in Sect. 4.4, it is then easy to extend ψ into the formula φ_f defined in (1) and submit it to an SMT solver such as Z3 to check its satisfiability.

The main challenge in using symbolic execution is to deal effectively with *path explosion*. Consecutive branches in a program can and, in practice, do increase the number of paths exponentially, especially when the program contains loops or recursive function calls. Symbolic execution engines implement various strategies to work around this issue, but these are beyond the scope of this paper. We refer the interested reader to the excellent survey by Baldoni et al. [8] for a general discussion of the topic, and to the KeY book [4] for concrete strategies implemented in the KeY symbolic execution engine used by *Minion*. We will briefly return to the question of how *Minion* deals with loops in Sect. 5.3.

5.2 Monitoring strategies

Our tool can monitor Java programs for both monolithic and distributed data minimality (MDM and DDM). We use the Java program from Fig. 2 in Sect. 4.1 as a running example to illustrate the use of *Minion* for monitoring MDM and DDM. Consider the method `fee` of the class `Toll`. The method does not contain any problematic control-flow constructs, such as loops, recursion or exceptions, and always returns a result. We can therefore safely treat `fee` as a (total) function. When running *Minion* on the method `fee`, the tool first builds the symbolic execution tree of the method using the KeY API and translates it into a logical specification suitable for dispatch to an SMT solver. This specification corresponds to the predicate $\varphi_{\texttt{fee}}$ defined in Sect. 4.4. Then, the monitor reads and parses traces in CSV format from an input file or standard input. Whenever *Minion* parses a new trace, it rechecks the entire set of traces read thus far for violation. In this way, the tool supports both online and offline monitoring. The number and format of the inputs in a trace is determined automatically from the method signature. Figure 3a shows example traces for the `fee` method. Columns 1–4 correspond to the parameters `h1`, `h2`, `h3` and `p`, respectively, while column 5 contains the result computed by `fee` for the given values.

By default, *Minion* monitors traces for DDM. Thus, when processing the traces given in Fig. 3a, it signals a violation after reading the second line because $\texttt{fee}(20, h_2, h_3, p) = \texttt{fee}(2, h_2, h_3, p)$ irrespective of the choice of h_2, h_3, and p. In contrast, all traces listed in Fig. 3b are accepted by *Minion* since they have been preprocessed by a distributed minimizer. Alternatively, *Minion* can be instructed to monitor traces for MDM in which case a violation is signaled when processing the last line of Fig. 3b, whereas all traces in Fig. 3c are accepted.

(a) unprocessed	(b) distributed minimal	(c) monolithic minimal
`20, 22, 1, 1, 770`	`0, 0, 0, 1, 770`	`20, 22, 1, 1, 770`
`2 , 2, 3, 5, 616`	`0, 0, 0, 3, 616`	`2, 2, 3, 5, 616`
`9 , 10, 10, 4, 792`	`9, 9, 9, 3, 792`	`9, 10, 10, 4, 792`
`23, 0, 2, 5, 616`	`0, 0, 0, 3, 616`	`2, 2, 3, 5, 616`
`10, 11, 14, 1, 990`	`9, 9, 9, 1, 990`	`14, 15, 15, 2, 990`
`8, 10, 11, 1, 990`	`0, 9, 9, 1, 990`	`14, 15, 15, 2, 990`
`...`	`...`	`...`

Fig. 3 Raw and minimized traces generated from `Toll.java`

5.2.1 Lazy vs. eager monitoring

Perhaps surprisingly, there are cases where *Minion* will detect a violation of DDM whereas it will not detect a violation of MDM. Consider the function $f(x, y) = x$. Since f simply ignores its second argument, it is clearly neither distributed nor monolithic minimal. When monitoring the pair of traces $(1, 2, 1)$ and $(3, 4, 3)$ for DDM, *Minion* detects a violation because $f(x, 2) = f(x, 4)$ for any choice of x. Note, however, that this situation does not appear among the observed traces since the two values for y in the respective traces differ. The tool reports a violation because a common value for x is found by our oracle when monitoring for DDM. When monitoring for MDM *Minion* does not detect the violation, because in this case there is no need to invoke the oracle.

Whether or not this is the intended behavior of the monitor depends on the assumption of whether the traces are collected from a program f or from the *combined* program $f \circ p$ (p being a minimizer). In the latter case, some combinations of inputs may never be observed as the inputs have been minimized. On the other hand, if traces are not considered preprocessed, we may wish to explore the behavior of f more exhaustively. For this purpose, *Minion* can be instructed to monitor a set of traces *eagerly* for MDM, resp. *lazily* for DDM. For the former, *Minion* considers not just the observed traces, but any combination of observed input values—even if that combination does not actually correspond to an observed trace. For the latter, *Minion* only considers combinations of inputs originating from traces with the same result value. For example, for the pair of input traces $(1, 2, 1)$ and $(3, 4, 3)$, *Minion* is able to find a violation in eager MDM mode since $f(1, 2) = f(1, 4)$, but not in lazy DDM mode since $f(1, 2) \neq f(3, 4)$.

5.3 Loops and loop invariants

Thus far, we have only considered simple programs without computational effects (such as exceptions or mutable state) and whose control flow does not include loops or recursive calls. These programs could safely be treated as (total) functions. Monitoring programs with loops for data minimality is more challenging, both conceptually and practically, because such programs can, in principle, be partial (i.e. non-terminating). To ensure correctness, we require all programs to be terminating and free from global side effects (though they may use e.g. exceptions and mutual state locally).

Minion provides two strategies for dealing with loops:

1. obtain an approximate logical characterization by *unrolling loops* to a fixed depth;
2. annotate loops with *loop invariants*, which allows KeY's symbolic execution engine to give a complete characterization of the method.

```
1  class Div {
2    //@ requires x >= 0 && y > 0;
3    //@ ensures (\result * y <= x) && (\result * y < x + y);
4    int posDiv(int x, int y) {
5      int q = 0;
6      //@ maintaining (r >= 0) && (r + q * y == x);
7      //@ decreasing r;
8      for (int r = x; r >= y; ++q) { r -= y; }
9      return q;
10   }
11 }
```

Fig. 4 A naive division algorithm for positive integers

The first strategy is suitable for loops with a bounded number of iterations. Intuitively, loop unrolling (also known as *loop unwinding*) replaces a loop with m copies of its loop body, where m is a fixed number called the unrolling depth. If the number of loop iterations is unbounded, this method results in an approximate specification by treating the program as partial: program executions exceeding the maximum number of iterations m are considered as returning an undefined result. The second option (loop invariants) is more flexible but also requires more work by the programmer who needs to specify logical invariants that hold during and after the execution of the loop. The KeY symbolic execution engine supports loop invariants specified in the *Java Modeling Language* (JML) [29] to verify termination and generate a logical specification for the program state during and after the loop [4]. This specification can then be used to characterize the overall result of the program precisely.

Consider, for example, the method `posDiv` given in Fig. 4, which implements naive integer division by repeatedly subtracting `y` from `x` and counting how many times this is possible. Our simple symbolic tree method cannot extract a complete logical characterization of `posDiv` automatically. Instead, one of the aforementioned strategies must be used. In the first case, choosing a large unrolling depth leads to high symbolic execution times because many copies of the loop body are generated and analyzed. A low unrolling depth, on the other hand, may affect accuracy. If the concrete values for `x` and `y` read from an input trace results in a number of loop iterations n below the unroll depth m, the resulting logical characterization is exact but if $n > m$ the characterization becomes an over-approximation and the monitor may fail to detect non-minimal traces. Indeed, it may not even detect inconsistent traces, i.e. traces where the expected output differs from the actual output computed at run time.

These false negatives can be avoided by annotating loops with JML loop invariants, as shown in Fig. 4. This invariant specifies that at every loop iteration the remainder r is positive and, when added to the iteration counter q times the divisor y, equals the original dividend x. With this invariant, the symbolic execution terminates quickly and without cutting off any branches, and *Minion* is able to extract a logical characterization for `posDiv`, which asserts that the eventual result q of the method must satisfy the equation $qy = x - r$ for some r such that $0 \leq r < y$. This is sufficient to correctly monitor any traces generated from `posDiv` for violation of DDM.

For an in-depth discussion about the effective use of loop unrolling and loop invariants, we refer the interested reader to the KeY book [4, Sect. 3.7.2].

5.4 Empirical evaluation

We evaluated *Minion* on a small set of programs to compare the output and relative run-time performance of its eager and lazy modes. Note that our primary goal in implementing *Minion*

Table 1 Mean running times and verdicts of *Minion* monitoring four different Java methods

	Symb. exec. (s)	K1: random (s) Eager	Lazy	E	L	K2: DDMin (s) Eager	Lazy	E	L	K3: MDMin (s) Eager	Lazy	E	L
T1	30.9 ±1.5	0.6 ±0.2	0.6 ±0.1	$\bot$	$\bot$	51.2 ±1.2	30.3 ±7.2	?	?	0.9 ±0.8	5.4 ±0.2	$\bot$	?
T2	9.2 ±0.7	0.4 ±0.1	0.4 ±0.0	$\bot$	$\bot$	17.3 ±0.8	13.9 ±2.2	?	?	17.3 ±0.7	3.7 ±0.3	?	?
CA	1.8 ±0.1	0.5 ±0.2	0.4 ±0.2	$\bot$	$\bot$	19.2 ±2.0	14.8 ±1.7	?	?	13.6 ±5.4	3.6 ±0.1	$\bot$?	?
LA	3.4 ±0.4	0.4 ±0.2	0.3 ±0.0	$\bot$	$\bot$					136.6 ±37.1	3.7 ±0.3	?	?

was to demonstrate the feasibility of our approach for gray-box monitoring. No attempt was made to optimize the performance of *Minion* and, accordingly, the goal of our evaluation is not to demonstrate scalability of our tool nor to establish a performance benchmark. Rather, the evaluation is meant to illustrate the behavior of the two DDM monitoring strategies (eager vs. lazy) under different workloads. These workloads have been chosen to deliberately exacerbate the differences between the two strategies.

For our evaluation, we ran *Minion* on four Java methods: the `fee` method from Fig. 2 (T1), a variant of that method that computes the fee on a road with only two toll stations instead of three (T2), as well as the `CreditApp` (CA) and `LoyaltyApp` (LA) programs introduced in [6]. Each method was monitored for DDM violation using three kinds of input traces:

(K1) random input values that respect the input specifications of the methods;
(K2) traces from (K1) minimized using a *distributed data minimizer* (DDMin);
(K3) traces from (K1) minimized using a *monolithic data minimizer* (MDMin).

The traces shown in Fig. 3 are subsets of the inputs generated for T1. We generated 10 instances of each kind, accounting for a total of 30 trace sets, each containing exactly 100 traces. All experiments were performed on a MacBook Pro with a 3.1 GHz Intel Core i5 processor and 16 GB of memory, running macOS 10.14. The results are summarized in Table 1.

Table 1 shows the mean running time and standard deviation in seconds, as well as the verdicts produced by *Minion*. The second column of the table reports the initial time spent on symbolic execution before any traces are processed. T1 incurs higher running times because T1 features several multiply-nested branches. The remaining columns report the execution times and verdict of the actual monitor.

The performance of eager and lazy monitoring is similar on random (K1) inputs because all cases have small (finite) input and output domains. Both approaches detect violations after processing only a few traces since even a small number of random traces cover a substantial part of the I/O domains, including those cases where violations occur.

As expected, the verdicts for the DDMin (K2) traces are inconclusive since DDM is not monitorable for satisfiability in general (by Lemma 5). Lazy monitoring does consistently better than eager monitoring on DDMin (K2) inputs because the eager strategy checks more input combinations before it (inevitably) comes to the same conclusion as the lazy strategy. The differences are relatively small for T1, T2 and CA because the ranges of these methods are small (< 10 elements). There is a bigger difference for LA, where the range is larger.

The performance of lazy monitoring on MDMin traces (K3) is consistently better than on DDMin traces (K2) because lazy DDM monitoring and MDM monitoring coincide for MDMin traces (no SMT invocations are necessary). On the other hand, the performance of eager monitoring for MDMin traces may change drastically depending on whether or not

the traces are also DDMin (which need not be the case). If they are, then eager monitoring has the same performance for MDMin traces as for DDMin traces. If they are not, the eager monitor might detect a violation early in the input set, cutting the overall execution time.

6 Related work

In this section, we review the related literature. The most relevant work are arguably the approaches developed for monitoring hyperproperties [3,17,24]. While we will compare and contrast our approach with these papers in detail later in this section, the main difference is that those techniques evaluate a formula based only on the current observation, providing an answer under the hypothesis that the observation is the full behavior of the system. The approach we propose in this paper, on the other hand, reasons not only about the executions observed so far, but also about other potential executions of the system that are not necessarily observed at run time.

6.1 LTL and monitorability for trace logics

Pnueli and Zaks [38] introduced the concept of monitorability for an LTL formula φ after observing a prefix O as the existence of an extension of O that permanently satisfies or permanently violates φ. It is known that the set of monitorable LTL properties is a superset of the union of safety and co-safety properties [12,13] and that it is also a superset of the set of obligation properties [22,23]. More recently, Havelund and Peled [26] introduced a finer-grained taxonomy distinguishing between *always* finitely satisfiable (resp. refutable), and *sometimes* finitely satisfiable (resp. refutable) where only some prefixes are required to be monitorable (for satisfaction). Their taxonomy also describes the relation between monitorability and classical safety properties. This more fine-grained distinction adds a new dimension to the monitorability cube in Fig. 1 which we will study in the future.

The recent work by Aceto et al. [2] builds a framework comparing the different notions of monitorability considered in the literature. For example, the original work by Pnueli and Zaks can be instantiated to mean that a property is monitorable when there is an observation O for which there is hope to monitor (that, is $\text{PZ}(O)$). This notion is called $\exists\text{PZ}(O)$. Dually, one can consider a $\forall\text{PZ}(O)$ which requires that there is hope to monitor the property for every observation O. Aceto et al. also study the existence of monitors for a variation of the fix-point logic RECHML that subsumes LTL as well as for a branching time variation [1,2].

While all the notions mentioned above ignore the system, predictive monitoring [43] considers the traces allowed in a given finite state system. None of the work mentioned in this subsection considers the trace/hyper and the computability dimensions of the monitorability cube in Fig. 1.

6.2 Monitoring hyperproperties

Previous techniques proposed for monitoring hyperproperties either (1) are limited to fragments for which definite verdicts can be given, or (2) given a finite set of finite traces M and a HyperLTL formula φ compute whether or not M satisfies φ for a finite length semantics of the temporal sub-formula. Note that for (2) future observations of new traces or extensions of observed traces may change the computed verdict. In the following, we refer to this notion

as *snapshot monitoring* to distinguish it from monitoring with respect to possible extended observations as studied in this paper.

Monitoring hyperproperties was first studied by Agrawal and Bonakdarpour [3] who provided the first notion of monitorability for HyperLTL (generalizing [38]) and gave an algorithm for a fragment of alternation-free HyperLTL. The monitoring algorithm only covers thes fragment of k-safety HyperLTL formulas, where the relation between traces does not involve temporal operators. Brett et al. later generalized the work in [3] to the full fragment of alternation-free formulas using formula rewriting [17], which can also monitor alternating formulas but again for snapshot monitoring (i.e. the verdict is computed with respect to a fixed finite set of finite traces).

A more general approach has been proposed by Finkbeiner et al. [24] who provide an automata-based algorithm for monitoring HyperLTL for a given trace set, enabling the computation of a verdict even for alternating formulas (still for snapshot monitoring). More specifically, they show that deciding monitorability (the definition given in [3]) for alternation-free HyperLTL is PSPACE-complete while the problem is undecidable in general. The monitoring algorithm generates an automaton where transitions are annotated by the trace variables in the input HyperLTL formula. They also study the role of certain formulas in achieving efficient monitoring algorithms, for example exploiting symmetries in the formula.

The complexity of (snapshot) monitoring different fragments of HyperLTL was studied in detail by Bonakdarpour and Finkbeiner [14] for Kripke structures with specific topologies, like trees, acyclic graphs, etc.

The idea of gray-box monitoring for hyperproperties—with the help of static information about the system under study—as a means for handling non-monitorable formulas, was first proposed by Bonakdarpour et al. [15]. In that paper it was also suggested the potential to incorporate abstract interpretation, symbolic execution, etc., in order to reason about quantifier alternation in HyperLTL formulas at run time. Our paper explores in detail one of these potential approaches by employing symbolic execution to extract a model from the source code, and an SMT-solver to reason about alternating HyperLTL formulas.

When one focuses on classical monitorability (where verdicts are permanent as opposed to changing verdicts as in snapshot monitoring) much less in known for HyperLTL, compared to LTL [26]. The results in Sect. 3 of this paper are a step in the direction of providing a similarly fine-grained landscape of monitorability concepts for HyperLTL.

6.3 Data minimization

A formal definition of *data minimization* in terms of strong dependency and derived concepts, and the concept of a *data minimizer* defined as a preprocessor to the data processor were first proposed by Antignac et al. [7]. In that paper, the authors considered both the monolithic and two versions of the distributed cases (including the weaker version considered in our paper), and they provided a proof-of-concept implementation to obtain data minimizers for a given program. The latter only works in practice provided enough annotations in the form of pre/post-conditions and invariants are provided. The approach in that paper is semantics-based, so finding a distributed minimizer is undecidable in general.

Formal and rigorous approaches to privacy have been advocated for some time (e.g. [42]), but the data minimization principle has not been precisely defined in the past except for the aforementioned paper. The main reason for this lack of work in the area is that it is a very hard problem. Indeed, the data minimization principle refers to both data at *collection* time and at *processing* time, as well as what is the explicit consent given by the data subject for

which specific purposes. There currently is no specification language to correctly formalize all these aspects of data minimization, let alone to verify or enforce them. The best that can be done today is to consider some aspects of the definition and address it gradually. The state of the art are the papers already mentioned ([7,36]) and some recent initial attempts to formalize the notion of purpose (e.g. [11]).

A closely related work is the notion of *minimal exposure* [5], which consists in performing a preprocessing of information on the client's side to give only the data needed to benefit from a service. The concept of data minimality is also closely related to information flow [20].

Equivalence partitioning by using symbolic execution was first introduced for test case generation by Richardson and Clarke [39], and later used by the *KeY* theorem prover [4]. Symbolic execution has limitations, especially when it comes to handling loops. Though being a main concern in theory and for some applications, *while* loops do not seem to be as widespread as `for` loops in practice. For instance, Malacaria et al. have been able to perform a symbolic execution-based verification of non-interference security properties from the C source of the real world *OpenSSL* library [31].

In this paper, we have focused on a weaker version of DDM which is not semantically black-box monitorable in general. For stronger versions (cf. [7]) it was shown by Pinisetty et al. [36] that the property is not (semantically) monitorable for satisfaction in general, but it is for violation. Their paper also discusses the runtime verification problem for other similar safety hyperproperties in the context of deterministic programs.

7 Conclusion and future work

In this paper, we have addressed the issue of monitorability with four main contributions. First, we have rigorously investigated the notion of monitorability, providing a new definition that is more general than previous definitions in the literature. Our definition considers the following dimensions: (1) a distinction between black-box and gray-box monitoring, (2) the nature of the property: trace properties or hyperproperties, (3) computability aspects of the monitor as a program. Second, we have shown that many hyperproperties that involve quantifier alternation are non-monitorable in a black-box manner and proposed a technique that allows, in certain cases, to monitor such properties by considering the use of oracles at run time. Such oracles are essentially static verifiers that may be called to further assist the runtime monitor in its decision on whether the property is violated or not. Third, we have considered a privacy property known to be non-monitorable using a black-box approach, and shown that we can use our gray-box technique to enable monitorability for violations of the property. The property under consideration is distributed data minimality (DDM), which may be expressed in HyperLTL involving one quantifier alternation. Our methodology to monitor violations of DDM is based on a model extracted from the program being monitored in the form of its symbolic execution tree, and the use of an SMT solver as an oracle. Finally, we have implemented a tool (*Minion*) and applied it to a number of representative examples to assess the feasibility of our approach.

Our work is a first step towards more ambitious tasks. First, since we have to model data we presented a variation of HyperLTL that includes relational predicate symbols. A full study, including the comparison of the expressive power of relational HyperLTL versus the conventional HyperLTL that only considers monadic predicate symbols, is an interesting theoretical problem for future research. Second, the non-monitorability results consider the simplest temporal formulas (safety). Future work includes capturing the conditions under

which other temporal formulas are non-monitorable. Another direction is to extend the proposed methodology for other hyperproperties that consider traces not from the same system, particularly in the concurrent and distributed setting. Indeed, most properties about concurrent and distributed systems are hyperproperties, and we would like to explore how to use our technique in the absence of total information when only having a local view (e.g. when only having access to the execution of a subset of the number of processors on a multi-core system).

In our proof-of-concept implementation, we use a combination of symbolic execution and an SMT solver as an oracle. In this case, we assume to have access to the source code of the program being monitored (to extract the symbolic execution tree), but that may not always be possible. As an alternative, we plan to explore the possibility of using, for instance, a model checker over a model of the system. We could also use bounded model checking as our verifier at run time by combining over- and under-approximated methods to deal with universal and existential quantifiers in HyperLTL formulas. Another interesting problem is to apply gray-box monitoring for hyperproperties with real-valued signals (e.g. HyperSTL [34]). Finally, we would like to extend the definition of (distributed) data minimality over reactive systems, and study what monitorability means in such a setting. Our preliminary study in this domain shows that having the right definition for reactive DDM is quite challenging.

Funding Open Access funding provided by University of Gothenburg.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Aceto L, Achilleos A, Francalanza A, Ingólfsdóttir A, Lehtinen K (2019a) Adventures in monitorability: from branching to linear time and back again. Proc ACM Program Lang (POPL'19) 3:52:1–52:29. https://doi.org/10.1145/3290365
2. Aceto L, Achilleos A, Francalanza A, Ingólfsdóttir A, Lehtinen K (2019b) An operational guide to monitorability. In: Proceedings of the 17th international conference on software engineering and formal methods (SEFM'19) vol 11724. Springer, LNCS,, pp 433–453. https://doi.org/10.1007/978-3-030-30446-1_23
3. Agrawal S, Bonakdarpour B (2016) Runtime verification of k-safety hyperproperties in HyperLTL. In: Proceedings of the IEEE 29th Computer Security Foundations (CSF'16). IEEE CS Press, pp 239–252. https://doi.org/10.1109/CSF.2016.24
4. Ahrendt W, Beckert B, Bubel R, Hähnle R, Schmitt PH, Ulbrich M (eds) (2016) Deductive software verification–the KeY book–from theory to practice, vol 10001. LNCS. Springer, Berlin. https://doi.org/10.1007/978-3-319-49812-6
5. Anciaux N, Nguyen B, Vazirgiannis M (2012) Limiting data collection in application forms: a real-case application of a founding privacy principle. In: Tenth annual international conference on privacy, security and trust (PST'12). IEEE, pp 59–66. https://doi.org/10.1109/PST.2012.6297920
6. Antignac T, Sands D, Schneider G (2016) Data minimisation: a language-based approach (long version). CoRR arXiv:1611.05642
7. Antignac T, Sands D, Schneider G (2017) Data minimisation: a language-based approach. In: Proceedings of the 32nd IFIP TC 11 international conference on ICT systems security and privacy protection (SEC'17), IFIPAICT, vol 502. Springer, pp 442–456. https://doi.org/10.1007/978-3-319-58469-0_30

8. Baldoni R, Coppa E, D'elia DC, Demetrescu C, Finocchi I (2018) A survey of symbolic execution techniques. ACM Comput Surv. https://doi.org/10.1145/3182657
9. Bartocci E, Falcone Y (eds) (2018) Lectures on runtime verification—introductory and advanced topics, vol 10457. LNCS. Springer, Berlin. https://doi.org/10.1007/978-3-319-75632-5
10. Bartocci E, Falcone Y, Francalanza A, Reger G (2018) Lectures on runtime verification, LNCS, vol 10457. Springer, Chap Introduction to runtime verification, pp 1–33. https://doi.org/10.1007/978-3-319-75632-5
11. Basin DA, Debois S, Hildebrandt TT (2018) On purpose and by necessity: compliance under the GDPR. In: Meiklejohn S, Sako K (eds) Financial cryptography and data security, LNCS, vol 10957. Springer, pp 20–37. https://doi.org/10.1007/978-3-662-58387-6_2
12. Bauer A, Leucker M, Schallhart C (2007) The good, the bad, and the ugly—but how ugly is ugly? In: Proceedings of the 7th international workshop on runtime verification (RV'07), LNCS, vol 4839. Springer, pp 126–138. https://doi.org/10.1007/978-3-540-77395-5_11
13. Bauer A, Leucker M, Schallhart C (2011) Runtime verification for LTL and TLTL. ACM T Softw Eng Methodol 20(4):14. https://doi.org/10.1145/2000799.2000800
14. Bonakdarpour B, Finkbeiner B (2018) The complexity of monitoring hyperproperties. In: Proceedings of the IEEE 31st computer security foundations symposium (CSF'18). IEEE, pp 162–174. https://doi.org/10.1109/CSF.2018.00019
15. Bonakdarpour B, Sánchez C, Schneider G (2018) Monitoring hyperproperties by combining static analysis and runtime verification. In: Proceedings of the 8th international symposium on leveraging applications of formal methods, verification and validation (ISoLA'18), Part II, LNCS, vol 11245. Springer, pp 8–27. https://doi.org/10.1007/978-3-030-03421-4_2
16. Boyer RS, Elspas B, Levitt KN (1975) SELECT—a formal system for testing and debugging programs by symbolic execution. In: Proceedings of the international conference on reliable software. ACM, pp 234–245. https://doi.org/10.1145/800027.808445
17. Brett N, Siddique U, Bonakdarpour B (2017) Rewriting-based runtime verification for alternation-free HyperLTL. In: Proceedings of the 23rd international conference on tools and algorithms for the construction and analysis of systems (TACAS'17), LNCS, vol 10206. Springer, pp 77–93. https://doi.org/10.1007/978-3-662-54580-5_5
18. Clarkson MR, Schneider FB (2010) Hyperproperties. J Comput Secur 18(6):1157–1210. https://doi.org/10.3233/JCS-2009-0393
19. Clarkson MR, Finkbeiner B, Koleini M, Micinski KK, Rabe MN, Sánchez C (2014) Temporal logics for hyperproperties. In: Proceedings of the third international conference on principles of security and trust (POST'14), LNCS, vol 8414. Springer, pp 265–284. https://doi.org/10.1007/978-3-642-54792-8_15
20. Cohen E (1977) Information transmission in computational systems. SIGOPS Oper Syst Rev 11(5):133–139. https://doi.org/10.1145/1067625.806556
21. European Parliament, Council of the European Union (2016) Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 april 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing directive 95/46/EC (General Data Protection Regulation). Offic J Eur Union L(119):1–88
22. Falcone Y, Fernandez JC, Mounier L (2009) Runtime verification of safety-progress properties. In: Proceedings of the 9th international workshop on runtime verification (RV'09), LNCS, vol 5779. Springer, pp 40–59. https://doi.org/10.1007/978-3-642-04694-0_4
23. Falcone Y, Fernandez JC, Mounier L (2012) What can you verify and enforce at runtime? Int J Softw Tools Technol Transf (STTT) 14(3):349–382. https://doi.org/10.1007/s10009-011-0196-8
24. Finkbeiner B, Hahn C, Stenger M, Tentrup L (2017) Monitoring hyperproperties. In: Proceedings of the 17th international conference on runtime verification (RV'17), LNCS, vol 10548. Springer, pp 190–207. https://doi.org/10.1007/978-3-319-67531-2_12
25. Havelund K, Goldberg A (2005) Verify your runs. In: Proceedings of the First IFIP TC 2/WG 2.3 conference on verified software: theories, tools, experiments (VSTTE'05), LNCS, vol 4171. Springer, pp 374–383. https://doi.org/10.1007/978-3-540-69149-5_40
26. Havelund K, Peled D (2018) Runtime verification: from propositional to first-order temporal logic. In: Proceedings of the 18th international conference on runtime verification (RV'18), LNCS, vol 11237. Springer, pp 90–112. https://doi.org/10.1007/978-3-030-03769-7_7
27. KeY contributors (accessed 25 Feb 2020) The KeY project. https://www.key-project.org
28. King JC (1976) Symbolic execution and program testing. Commun ACM 19(7):385–394. https://doi.org/10.1145/360248.360252
29. Leavens GT, Poll E, Clifton C, Cheon Y, Ruby C, Cok DR, Müller P, Kiniry J, Chalin P, Zimmerman DM (2013) JML reference manual. Department of Computer Science, Iowa State University. http://www.jmlspecs.org

30. Leucker M, Schallhart C (2009) A brief account of runtime verification. J Log Algebr Program 78(5):293–303. https://doi.org/10.1016/j.jlap.2008.08.004
31. Malacaria P, Tautchning M, DiStefano D (2016) Information leakage analysis of complex C code and its application to OpenSSL. In: Proceedings of the 7th international symposium on leveraging applications of formal methods, verification and validation (ISoLA'16), Part I, LNCS, vol 9952. Springer, pp 909–925. https://doi.org/10.1007/978-3-319-47166-2_63
32. Microsoft Research (accessed 25 Feb 2020) The Z3 theorem prover. https://github.com/Z3Prover/z3
33. Moura LD, Bjørner N (2008) Z3: An efficient SMT solver. In: Proceedings of the 14th international conference on tools and algorithms for the construction and analysis of systems (TACAS'08), LNCS, vol 4963. Springer, Springer, pp 337–340, https://doi.org/10.1007/978-3-540-78800-3_24
34. Nguyen LV, Kapinski J, Jin X, Deshmukh JV, Johnson TT (2017) Hyperproperties of real-valued signals. In: Proceedings of the 15th ACM-IEEE international conference on formal methods and models for system design (MEMOCODE'17). ACM, pp 104–113. https://doi.org/10.1145/3127041.3127058
35. Pinisetty S, Antignac T, Sands D, Schneider G (2018) Monitoring data minimisation. CoRR arXiv:1801.02484
36. Pinisetty S, Sands D, Schneider G (2018) Runtime verification of hyperproperties for deterministic programs. In: Proceedings of the 6th conference on formal methods in software engineering (FormaliSE@ICSE'18). ACM, pp 20–29. https://doi.org/10.1145/3193992.3193995
37. Pnueli A (1977) The temporal logic of programs. In: Proceedings of the 18th IEEE symposium on foundations of computer science (FOCS'77). IEEE CS Press, pp 46–67. https://doi.org/10.1109/SFCS.1977.32
38. Pnueli A, Zaks A (2006) PSL model checking and run-time verification via testers. In: Proceedings of the 14th international symposium on formal methods (FM'06), LNCS, vol 4085. Springer, pp 573–586. https://doi.org/10.1007/11813040_38
39. Richardson DJ, Clarke LA (1981) A partition analysis method to increase program reliability. In: Proceedings of the 5th international conference on software engineering (ICSE'81). IEEE Press, Piscataway, pp 244–253
40. Sánchez C, Schneider G, Ahrendt W, Bartocci E, Bianculli D, Colombo C, Falcone Y, Francalanza A, Krstić S, Nickovic D, Pace GJ, Rufino J, Signoles J, Traytel D, Weiss A (2019) A survey of challenges for runtime verification from advanced application domains (beyond software). Form Methods Syst Des. https://doi.org/10.1007/s10703-019-00337-w
41. Stucki S, Sánchez C, Schneider G, Bonakdarpour B (2019) Gray-box monitoring of hyperproperties. In: Proceedings of the third world congress on formal methods (FM'19), LNCS, vol 11800. Springer, pp 406–424. https://doi.org/10.1007/978-3-030-30942-8_25
42. Tschantz MC, Wing JM (2009) In: Proceedings of the second world congress on formal methods (FM'09), LNCS. Springer, Berlin, Chap Formal methods for privacy, pp 1–15. https://doi.org/10.1007/978-3-642-05089-3_1
43. Zhang X, Leucker M, Dong W (2012) Runtime verification with predictive semantics. In: Proceedings of 4th NASA international symposium on formal methods (NFM'12), LNCS, vol 7226. Springer, pp 418–432. https://doi.org/10.1007/978-3-642-28891-3_37

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Affiliations

Sandro Stucki[1] · César Sánchez[2] · Gerardo Schneider[1] · Borzoo Bonakdarpour[3]

✉ Sandro Stucki
sandro.stucki@gu.se

César Sánchez
cesar.sanchez@imdea.org

Gerardo Schneider
gerardo@cse.gu.se

Borzoo Bonakdarpour
borzoo@msu.edu

[1] Department of Computer Science and Engineering, University of Gothenburg, 412 96 Gothenburg, Sweden

[2] IMDEA Software Institute, Campus de Montegancedo s/n, 28223 Pozuelo de Alarcon, Madrid, Spain

[3] Department of Computer Science and Engineering, Michigan State University, East Lansing, MI 48824, USA