Monitorability for the Modal Mu-Calculus over Systems with Data: From Practice to Theory

Luca Aceto
Dept. of Computer Science,
Reykjavik University, Iceland
Gran Sasso Science Institute, L'Aquila, Italy

Antonis Achilleos
Dept. of Computer Science,
Reykjavik University, Iceland

Duncan Paul Attard
University of Malta, Msida, Malta

Léo Exibard
LIGM, CNRS, Univ Gustave Eiffel,
F77454 Marne-la-Vallée, France

Adrian Francalanza
University of Malta, Msida, Malta

Anna Ingólfsdóttir
Dept. of Computer Science,
Reykjavik University, Iceland

Karoliina Lehtinen
CNRS, Aix-Marseille University, LIS,
Marseille, France

Abstract

Runtime verification consists in checking whether a system satisfies a given specification by observing the execution trace it produces. In the regular setting, the modal μ-calculus provides a versatile formalism for expressing specifications of the control flow of the system. This paper focuses on the *data* flow and studies an extension of that logic that allows it to express data-dependent properties, identifying fragments that can be verified at runtime and with what correctness guarantees. The logic studied here is closely related with register automata with *guessing*. That correspondence yields a monitor synthesis algorithm, and a strict hierarchy among the various fragments of the logic, in contrast to the regular setting. We then exhibit a fragment of the logic that can express all monitorable formulae in the logic without greatest fixed-points but not in the full logic, and show this is the best we can get.

2012 ACM Subject Classification Theory of computation → Logic and verification; Theory of computation → Modal and temporal logics; Theory of computation → Automata over infinite objects

Keywords and phrases Runtime verification, monitorability, μHML with data, register automata

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2025.4

Related Version *Full Version*: https://arxiv.org/abs/2506.06172

1 Introduction

Runtime verification is an increasingly important lightweight validation technique that consists in checking a specification by observing an execution trace at runtime [14]. Not all system properties can be verified this way, e.g. those that mention behaviours that are not observed in the given trace, or limit behaviours such as "every request is always eventually granted". However, it can check properties for which an exhaustive state-space exploration is impractical, and verify systems whose model is unavailable, e.g. closed source code.

In the classical setting, system properties are typically expressed through formalisms whose models are (ω-)words or (ω-)trees, e.g. linear-time temporal logic (LTL), computation tree logic (CTL/CTL*) or the modal μ-calculus (equivalently, Hennessy-Milner logic with recursion), all falling within the realm of (ω-)regular behaviours [32]. While this setting enjoys numerous desirable properties (reasonable computational complexity, closure properties,

36th International Conference on Concurrency Theory (CONCUR 2025).
Editors: Patricia Bouyer and Jaco van de Pol; Article No. 4; pp. 4:1–4:21
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

correspondence with automata models, etc.), it falls short of capturing properties of the *data flow* of the system – what information it manipulates and how – since the alphabet of the traces or computation trees is assumed to be finite and typically small, corresponding to a focus on the *control* flow of the system – which signals it emits and when. The data flow has typically higher complexity, due to its unbounded nature, making it seem out of reach. However, due to the ubiquity of data manipulation and the increasing availability of computational power, numerous formal methods have shifted the focus to data, in runtime verification [41, 43], model-checking [20] and reactive synthesis [31, 33].

In the field of runtime verification, tools supporting monitoring of data-dependent properties of systems have been available for some time and have been applied in a variety of settings [2, 8, 11–13, 15, 16, 23, 28, 40, 47] (see also the surveys [34, 43]). However, to our mind, the systematic development of their theoretical underpinnings has lagged behind their practice. To quote Milner in [48] "the design of computing systems can only properly succeed if it is well grounded in theory". That quote motivates us to study the theoretical foundations of runtime monitoring for properties of data-dependent systems and to provide a systematic analysis of which properties can be monitored at runtime and with what correctness guarantees, paralleling our analysis in the regular setting [4].

To represent systems with data, we use data words and trees, whose elements are pairs of a letter from a finite alphabet and a value from an infinite domain, structured by a set of predicates to compare data values. In [41], the authors introduce a modal μ-calculus based formalism to express data properties that can be monitored at runtime. In [2], we introduced a variant with only the equality predicate. We provide an in-depth study of this logic by studying its expressiveness and monitorability. This reveals an intricate landscape where, in contrast to the ω-regular setting, most variations on the notion of monitor are *not* equivalent, demonstrating the need for distinct monitor models dependent on the property being monitored for. We also uncover a mistake in [7, Theorem 18], since what was believed to be a normal form is actually less expressive than the full monitor model. This observation may trigger development in the corresponding tool. The main contributions of the paper are:

- A formulation of the Hennessy-Milner logic with recursion over data words (Sec. 2) which refines the one in [2]. It is inspired from [41], with the equality predicate only (and hence without functions).
- The delineation of the HML^d fragment, capturing all completely monitorable properties expressible over data domains with only equality (Sec. 3.2 and Theorem 14).
- The delineation of the cHML^d fragment, which we show is monitorable for satisfactions by a natural extension of the monitor model in [4], along with its compositional synthesis algorithm (Sec. 3.3 and Theorem 17). This monitor model is moreover as expressive as alternating register automata with existential guessing [35, 36] (Theorem 18).
- We establish that, contrary to the ω-regular setting [4], this fragment is not maximal (Proposition 21), and delineate a fragment (Sec. 4.1) that is maximal among properties without greatest fixed points (Theorem 26). We show that the latter fragment is however not maximal in general, and that there is no maximal monitorable fragment whose membership is decidable, in the sense that one could decide whether an input formula can be written in that fragment (Corollary 29). To our minds, those results are the most original contributions of the paper.

Due to space limitations, many proofs are omitted, but can be found in the full version on arXiv.

Related work. Runtime verification tools often integrate some data capabilities. Indeed, according to Falcone *et al.* [34], 13 of the 20 tools surveyed have some data in the input specification. Among tools with data support, we mention AspectJ [8], with data included in regular expression matching, the MOP Framework, which integrates runtime verification with data-handling capabilities into the software development cycle [47]. Rule-based monitor Ruler [13] and the corresponding logic Eagle [12] have both been extended with data parameters. The work [28] uses SMT solvers to handle data added to the (potentially infinite state) monitor directly. Trace slicing reduces the problem to checking projections of traces onto a finite set of values [23] while quantified event automata allow for initial quantification over the domain and then spawn copies of the automaton for all possible values [11]. Finally, the work in [42] studies how to efficiently handle the unbounded memory needs induced by the manipulation of data values in the Dejavu tool, using binary decision diagrams [22] among other techniques.

Another approach is to add data to the logic and monitor fragments thereof. The study in [15] proposes monitors for security policies expressed in metric first order temporal logic. Temporal Object Property Language is a high level logic designed for Java developers, with register automata as a backend formalism [40], bridging the programmer–automata gap.

On the theoretical side, in [16] Bauer et al. study the monitorability of LTL^{FO}, LTL with first order quantification over data. The prefix problem is undecidable, so there is no hope of computing complete monitors but the authors establish a hierarchy based on how much of the trace must be stored. Regarding specifications, the relations between the many logics and automata handling data [27] remain largely unmapped, and most models are not equivalent. Among automata models, register automata are well studied [18]. Pebble automata [49] are closer to logic, but at the cost of decidability. Class memory automata and data automata coincide [17]. Among logics, LTL has been extended to data domains in various ways [27]. In particular, *freeze* LTL is recognisable by alternating register automata [29], which are also closely related to an extension of the modal μ-calculus [46]. We also mention the Logic of Repeating Values [38] and first-order two-variable logic [19] which both have promising algorithmic properties. Beyond the equality predicate, some logics also handle richer domains such as uninterpreted functions [39], and $(\mathbb{Q}, <)$ – the latter is closely related to timed formalisms [37], where the infinity of the alphabet stems from real-valued timestamps [44].

2 The Logic $\mu\mathrm{HML}^d$

In this section, we define $\mu\mathrm{HML}^d$, an extension of μHML tailored to express properties of traces of system executions that contain data values. It is a reformulation of the one in [2], which improves it by bringing a more explicit and versatile handling of data through quantifiers and data valuations instead of bindings and substitutions. The former increases expressiveness by allowing one to *guess* data values, while the latter clarifies how data values interact with fixed points. Note that μHML comes in two flavours: *branching time* (the logic describes possible executions of the system) and *linear time* (it describes actual traces). Here, we are concerned with the linear-time setting.

2.1 Data Words and Traces

In formal methods, data words and trees constitute popular formalisms to model respectively the traces and possible executions of systems [51]. Since we consider linear-time properties, we model execution *traces* as data ω-words. They consist in infinite words whose elements are pairs of a letter from a finite alphabet and of a *data value* from an infinite domain. The finite alphabet plays no role here and can be simulated, so we omit it for simplicity [49]. A data word is thus an infinite sequence of values from an infinite domain.

For the rest of the paper, we fix a countably infinite *data domain* $\mathbb{D}$, whose only predicate is "=" and is decidable. An *action* is modelled as a data value $d \in \mathbb{D}$. An infinite (respectively, finite) *trace* is a data word, *i.e.,* an infinite (resp., finite) sequence $t \in \mathbb{D}^\omega$ (resp., $w \in \mathbb{D}^n$ for some $n \in \mathbb{N}$); the set of all infinite traces is denoted $\text{TRC} = \mathbb{D}^\omega$ (resp., $\text{FTRC} = \mathbb{D}^*$ for finite traces). For $w = w_0 \ldots w_n \in \text{FTRC}$ and $u = u_0 u_1 \cdots \in \text{FTRC} \cup \text{TRC}$, the *concatenation* of w and u is $w \cdot u = w_0 \ldots w_n u_0 \ldots$ (we may omit the $\cdot$). When $u = y \cdot v$, y is a *prefix* of u, and v is a *suffix* of u. The set of suffixes of u is denoted $\texttt{suffix}(u)$.

2.2 Syntax and Semantics

We define an extension of μHML, called μHML^d. Its syntax and semantics are described in Fig. 1 on page 5. Formulae are built from a countable set of formula variables, $X, Y \in \text{FVAR}$, and data variables, $x, y \in \text{DVAR}$, ranging over an infinite domain of data values, $d \in \mathbb{D}$. In addition to the standard Boolean constructs, μHML^d can express recursive properties as least ($\mathsf{min} X.(\varphi)$) and greatest ($\mathsf{max} X.(\varphi)$) fixed-point formulae that bind the free occurrences of X in φ. The logic includes the *possibility* ($\langle b \rangle \varphi$) and *necessity* ($[b]\varphi$) modal constructs. To reason on the data carried by process actions, modalities are augmented with decidable, quantifier-free Boolean *constraint expressions*, $b, c \in \text{BEXP}$, defined over $\mathbb{D}$ and $\text{DVAR} \cup \{\star\}$, where $\star \notin \text{DVAR}$ is a placeholder variable for the current action $d \in \mathbb{D}$. The free data variables $x \in \text{DVAR}$ that appear in b are bound by existential and universal quantification constructs $\exists \boldsymbol{x}.\varphi$ and $\forall \boldsymbol{x}.\varphi$.

In what follows, the standard notions of open and closed expressions, free and bound variables, and formula equivalence up to alpha-conversion are used. We assume wlog that every occurrence of each fixed-point variable is within the scope of a modal operator in its defining fixed-point formula. This is the case e.g. of $\mathsf{max} X.([b]X)$, but not of $\mathsf{max} X.(X \wedge [b]X)$.

We define the domains $\text{DENV} = \text{DVAR} \rightharpoonup \mathbb{D}$ of data environments, $\text{DINT} = \text{DENV} \rightharpoonup 2^{\text{TRC}}$ of data interpretations, and $\text{TENV} = \text{FVAR} \rightharpoonup \text{DINT}$ of trace environments (where $A \rightharpoonup B$ denotes the set of partial functions from set A to set B). A *data environment*, $\delta \in \text{DENV}$, is a partial function with a finite domain mapping data variables to values from $\mathbb{D}$; analogously, a *trace environment*, $\rho \in \text{TENV}$, maps formula variables to data interpretations $F, G \in \text{DINT}$, that given δ, return a set of traces, whose intended meaning is the interpretation of the formula variable in the data environment δ.

The *linear-time* semantics of μHML^d is given by the denotational semantic function, $[\![-]\!]$, defined inductively in Fig. 1. In $[\![-]\!]$, formulae are interpreted w.r.t. a trace environment ρ that gives meaning to formula variables, and a data environment δ that assigns values to data variables in Boolean constraint expressions. Expressions are of the form:

$$b, c \in \text{BEXP} ::= \mathsf{true} \;\mid\; e = f \;\mid\; \neg b \;\mid\; b \wedge c \tag{1}$$

where $e, f \in \text{DVAR} \sqcup \{\star\}$. An expression b defines a set of *external* system actions. An action d is in this set when the data value it carries satisfies b with regards to the data environment δ, *i.e.,* $b\delta[\star \mapsto d] \Downarrow \mathsf{true}$, where, for any function $f : A \to B$, any a (not necessarily in A) and any $b \in B$, $f[a \mapsto b]$ denotes the function that maps a to b and agrees with f over $A \backslash \{a\}$.

μHML^d Syntax. $\varphi,\psi \in \mu\text{HML}^d ::= \mathsf{tt} \mid \mathsf{ff} \mid \langle b\rangle\varphi \mid [b]\varphi \mid \exists \boldsymbol{x}.\varphi \mid \forall \boldsymbol{x}.\varphi \mid \varphi \vee \psi \mid \varphi \wedge \psi \mid \min X.(\varphi) \mid \max X.(\varphi) \mid X$

Fragments.

$$\varphi,\psi \in \text{cHML}^d ::= \mathsf{tt} \mid \langle b\rangle\varphi \mid \exists \boldsymbol{x}.\varphi \mid \varphi \vee \psi \mid \varphi \wedge \psi \mid \min X.(\varphi) \mid X$$

$$\varphi,\psi \in \text{sHML}^d ::= \mathsf{ff} \mid [b]\varphi \mid \forall \boldsymbol{x}.\varphi \mid \varphi \vee \psi \mid \varphi \wedge \psi \mid \max X.(\varphi) \mid X$$

$$\varphi,\psi \in \text{disjHML}^d ::= \mathsf{tt} \mid \langle b\rangle\varphi \mid \exists \boldsymbol{x}.\varphi \mid \varphi \vee \psi \mid \min X.(\varphi) \mid X$$

$$\varphi,\psi \in \text{HML}^d ::= \mathsf{tt} \mid \mathsf{ff} \mid \langle b\rangle\varphi \mid [b]\varphi \mid \exists \boldsymbol{x}.\varphi \mid \forall \boldsymbol{x}.\varphi \mid \varphi \vee \psi \mid \varphi \wedge \psi$$

Semantics.

$$[\![\mathsf{tt},\rho,\delta]\!] \triangleq \text{Trc} \qquad [\![\mathsf{ff},\rho,\delta]\!] \triangleq \varnothing \qquad [\![X,\rho,\delta]\!] \triangleq \big(\rho(X)\big)(\delta)$$

$$[\![\langle b\rangle\varphi,\rho,\delta]\!] \triangleq \big\{t \mid (\exists u,d.t = du \text{ and } b\delta[\star\mapsto d] \Downarrow \mathsf{true} \text{ and } u \in [\![\varphi,\rho,\delta]\!])\big\}$$

$$[\![[b]\varphi,\rho,\delta]\!] \triangleq \big\{t \mid (\forall u,d.(t = du \text{ and } b\delta[\star\mapsto d] \Downarrow \mathsf{true}) \text{ implies } u \in [\![\varphi,\rho,\delta]\!])\big\}$$

$$[\![\exists \boldsymbol{x}.\varphi,\rho,\delta]\!] \triangleq \bigcup_{d\in\mathbb{D}} [\![\varphi,\rho,\delta[x\mapsto d]]\!] \qquad [\![\forall \boldsymbol{x}.\varphi,\rho,\delta]\!] \triangleq \bigcap_{d\in\mathbb{D}} [\![\varphi,\rho,\delta[x\mapsto d]]\!]$$

$$[\![\varphi\vee\psi,\rho,\delta]\!] \triangleq [\![\varphi,\rho,\delta]\!] \cup [\![\psi,\rho,\delta]\!] \qquad [\![\varphi\wedge\psi,\rho,\delta]\!] \triangleq [\![\varphi,\rho,\delta]\!] \cap [\![\psi,\rho,\delta]\!]$$

$$[\![\min X.(\varphi),\rho,\delta]\!] \triangleq \Big(\bigsqcap\big\{F \mid \lambda\delta'.[\![\varphi,\rho[X\mapsto F],\delta']\!] \sqsubseteq F\big\}\Big)(\delta)$$

$$[\![\max X.(\varphi),\rho,\delta]\!] \triangleq \Big(\bigsqcup\big\{F \mid F \sqsubseteq \lambda\delta'.[\![\varphi,\rho[X\mapsto F],\delta']\!]\big\}\Big)(\delta)$$

Figure 1 Syntax and linear-time semantics of μHML^d.

Satisfaction of an expression is then defined as:

$$b\delta \triangleq \begin{cases} \mathsf{true} & b = \mathsf{true} \\ (e\delta) = (f\delta) & b = (e = f) \\ \neg(c\delta) & b = \neg c \\ (c'\delta)\wedge(c''\delta) & b = c' \wedge c'' \end{cases} \qquad e\delta \triangleq \begin{cases} \delta(x) & e = x \text{ and } x \in \mathsf{dom}(\delta) \\ x & e = x \text{ and } x \notin \mathsf{dom}(\delta) \end{cases} \tag{2}$$

Possibility formulae $\langle b\rangle\varphi$ denote all the traces $t = du$ that begin with an action d that is in the action set described by $b\delta$ *and* whose tail u satisfies the continuation formula φ. Dually, necessity formulae $[b]\varphi$ describe all the traces that, *whenever* they begin with such an action d, continue with a trace that satisfies φ. Note that in the linear-time setting, necessity can be expressed as possibility: $[b]\varphi \equiv \langle\neg b\rangle\mathsf{tt} \vee \langle b\rangle\varphi$, and dually $\langle b\rangle\varphi = [\neg b]\mathsf{ff} \wedge [b]\varphi$. The existential quantifier $\exists \boldsymbol{x}.\varphi$ is interpreted as the set of traces that satisfy φ by assigning *some* $d \in \mathbb{D}$ to x; the universal quantifier $\forall \boldsymbol{x}.\varphi$ is the set of traces satisfying φ under *all* such assignments. Formulae are only interpreted with regards to data environments whose domain includes the set of free data variables occurring in them. Note that existential quantification cannot be expressed using universal quantification, except using negation, which is not allowed in the syntax outside modalities.

Since the logic does not have an explicit negation operator, for all φ the semantic function $[\![\varphi,\rho,\delta]\!]$ is monotonic in ρ over the complete lattice $(\text{DInt},\sqsubseteq)$, where the partial order $\sqsubseteq$ corresponds to graph inclusion. Formally, it is defined, for all $F,G \in \text{DInt}$, as $F \sqsubseteq G$ whenever $\forall\delta \in \text{DEnv}.F(\delta) \subseteq G(\delta)$. As is standard in the modal μ-calculus, recursion is interpreted through fixed points: by the Knaster-Tarski theorem [53], $\min X.(\varphi)$ and $\max X.(\varphi)$ respectively correspond to the least and greatest fixed point of the operator that maps a data interpretation $F : \text{DEnv} \to 2^{\text{Trc}}$ to the data interpretation $\delta \mapsto [\![\varphi,\rho[X\mapsto F],\delta]\!]$. This is the analogue of the operator used to define the semantics of the modal μ-calculus over traces, lifted to the case of infinite alphabets by parameterising the interpretation by a data environment, in the spirit of [41]. To obtain the sought interpretation for $\mathsf{min}\, X.(\varphi)$ and $\mathsf{max}\, X.(\varphi)$, one then applies the least (resp., greatest) fixed point of this operator (which is a function from data environments to sets of traces) to the current data environment δ.

By construction, they both satisfy the following fixed-point equations where, for all formulae $\varphi,\psi \in \mu\mathrm{HML}^d$ and all recursion variables $X \in \mathrm{FVar}$, we write $\varphi[{}^{\psi}/_X]$ for the formula that results by the standard capture-avoiding substitution of all free occurrences of X in φ with ψ:

▶ **Proposition 1.** *For all formulae φ, all trace environments X, all data environments δ, $[\![\mathsf{min}\, X.(\varphi),\rho,\delta]\!] = [\![\varphi[{}^{\mathsf{min}\, X.(\varphi)}/_X],\rho,\delta]\!]$ and $[\![\mathsf{max}\, X.(\varphi),\rho,\delta]\!] = [\![\varphi[{}^{\mathsf{max}\, X.(\varphi)}/_X],\rho,\delta]\!]$.*

When a formula is closed with regards to recursion variables (respectively, data variables), its interpretation does not depend on the trace environment ρ (resp., the data environment δ) and we write $[\![\varphi,\delta]\!]$ (resp., $[\![\varphi,\rho]\!]$) in lieu of $[\![\varphi,\rho,\delta]\!]$. For closed formulae, we drop both and write $[\![\varphi]\!]$ in lieu of $[\![\varphi,\rho,\delta]\!]$ for clarity. We say that a trace t satisfies a closed formula φ if $t \in [\![\varphi]\!]$, and violates φ if $t \notin [\![\varphi]\!]$. In the following, in all closed formulae φ we assume that each recursion variable X appears in a unique fixed-point formula $\mathtt{fx}_\varphi(X)$, which is either of the form $\mathsf{min}\, X.(\varphi_X)$ or $\mathsf{max}\, X.(\varphi_X)$. If $\mathtt{fx}(X)$ is $\mathsf{min}\, X.(\varphi_X)$, then X is called an *lfp variable*; otherwise, X is called a *gfp variable*. We write $X \le Y$ when φ_X is a subformula of φ_Y, $X < Y$ when moreover $X \neq Y$, and denote by $\mathtt{sub}(\varphi)$ the set of subformulae of φ.

▶ **Example 2.** To give an intuition of the logic and its expressiveness, here are a few elementary $\mu\mathrm{HML}^d$ properties, along with their respective fragments:

- The first and second data values are equal (HML^d):

$$\varphi_3 \triangleq \exists \boldsymbol{x}.\langle x = \star\rangle\langle x = \star\rangle \mathsf{tt} \tag{3}$$

  Indeed, the only way for the first modality $\langle x = \star\rangle$ to be satisfied is if x takes the value of the first data value. Then, the second modality $\langle x = \star\rangle$ is satisfied iff the second value is equal to x, hence to the first value.
- The first data value appears again ($\mathrm{DISJHML}^d$):

$$\varphi_{\mathrm{leak}} \triangleq \exists \boldsymbol{x}.\langle x = \star\rangle \mathsf{min}\, X.(\langle x = \star\rangle \mathsf{tt} \vee \langle x \neq \star\rangle X) \tag{4}$$

  where we use $x \neq \star$ to abbreviate $\neg(x = \star)$. As above, x stores the first data value. Then, we use recursion to look for the second occurrence. Intuitively, on encountering a fixed-point variable X the formula recurses, i.e. we can replace X with the whole $\mathsf{min}X.(\varphi)$ that encloses it, as expressed by Proposition 1. Here, the formula recurses while it encounters values satisfying $x \neq \star$, and is satisfied (reaching tt) if it encounters a value satisfying $x = \star$, viz. the first value in the trace. Since this is a least fixed point (min), the formula is violated if it recurses ad infinitum, i.e. if the first value never appears again.
- *Some* data value appears at least twice ($\mathrm{DISJHML}^d$):

$$\varphi_5 \triangleq \exists \boldsymbol{x}.\mathsf{min}\, X.(\langle x = \star\rangle \mathsf{min}\, Y.(\langle x \neq \star\rangle X \vee \langle x = \star\rangle \mathsf{tt} \vee \langle x \neq \star\rangle Y)) \tag{5}$$

  For a given value of x, the formula accepts only if this value is found once (first disjunct of the first min) and then again (first disjunct of the second, nested min). Overall, the formula accepts whenever there exists such a value, which thus appears twice.
- All data values are pairwise distinct (negation by dualisation of a $\mathrm{DISJHML}^d$ formula):

$$\varphi_6 \triangleq \forall \boldsymbol{x}.\mathsf{max}\, X.([x = \star]\mathsf{max}\, Y.([x \neq \star]X \wedge [x = \star]\mathsf{ff} \wedge [x \neq \star]Y)) \tag{6}$$

  Dually to the above one, this formula *rejects* whenever some value appears twice.

- The first data value eventually repeats, and in between all data values are pairwise distinct (cHML^d):

$$\varphi_{\text{dist}} \triangleq \exists \boldsymbol{x}.\langle\star = x\rangle \mathsf{min}\, X.\Big(\langle x = \star\rangle \mathsf{tt} \vee \big(\exists \boldsymbol{y}.\langle\star = y\rangle \mathsf{min}\, Y.(\langle\star = x\rangle \mathsf{tt} \vee \langle\star \neq x \wedge \star \neq y\rangle Y)\big) \wedge \langle\star \neq x\rangle X\Big)$$

Here, the first diamond $\langle.\rangle$ implies that x is bound to the first data value. Then, the first min is satisfied when x occurs again, or when the rhs of the first disjunction is satisfied. This happens when the current data value (bound to y thanks to the $\langle\star = y\rangle$ diamond) does not appear before x is found, as checked by the $\mathsf{min}.Y$, *and* that the overall property is true at next step.
- There exists a data value that never appears (μHML^d):

$$\varphi_7 \triangleq \exists \boldsymbol{x}.\mathsf{max}\, X.([x = \star]\mathsf{ff} \wedge [x \neq \star]X) \tag{7}$$

As for φ_6, the max allows one to forbid a data value (existentially guessed using the $\exists$ quantifier) from appearing in a trace.

2.3 Satisfiability and Validity

Over data words, the infinity of the domain implies that compromises have to be made between expressiveness, closure properties and decidability [17, 27]. By adapting the classical encoding [25, Section 12], one can observe that μHML^d is strictly more expressive than LTL with freeze [30]. Thus, in our setting, decidability fails: the satisfiability and validity problems of μHML^d are undecidable, in contrast with the finite alphabet case (μHML) [54]. By adapting the reduction of [49, Theorem 18], we can sharpen the undecidability result:

▶ **Theorem 3.** *The validity problem for* disjHML^d *is undecidable.*

▶ **Theorem 4.** *The satisfiability problem for* cHML^d *is undecidable.*

The decidability picture for μHML^d is quite grim, but fortunately, as we will see in Sec. 3, this does not prevent us from delineating monitorable fragments of that logic.

2.4 Annotation Semantics

We introduce an alternative semantics for formulae in μHML^d, to better argue about the monitorability of a formula. Annotations are analogous to choice functions [52, Section 4] (see also [24, Theorem 2.1]), and consist in (possibly infinite) witnesses that a formula holds.

▶ **Definition 5.** *An* annotation *is a graph* $(A, \rightarrowtail)$*, where* $A \subseteq \mu\text{HML}^d \times \text{DEnv} \times \text{Trc}$*, and:*

- *it is not the case that* $(\mathsf{ff}, \delta, t) \in A$ *for any* $t \in \text{Trc}$ *and* $\delta \in \text{DEnv}$*;*
- *if* $(\langle b\rangle\varphi, \delta, dt) \in A$*, then* $b\delta[\star \mapsto d] \Downarrow \mathsf{true}$*,* $(\varphi, \delta, t) \in A$*, and* $(\langle b\rangle\varphi, \delta, dt) \rightarrowtail (\varphi, \delta, t)$*;*
- *if* $([b]\varphi, \delta, dt) \in A$*, and* $b\delta[\star \mapsto d] \Downarrow \mathsf{true}$*, then* $(\varphi, \delta, t) \in A$*, and* $([b]\varphi, \delta, dt) \rightarrowtail (\varphi, \delta, t)$*;*
- *if* $(\exists \boldsymbol{x}.\varphi, \delta, t) \in A$*, then* $(\varphi, \delta[x \mapsto d], t) \in A$ *and* $(\exists \boldsymbol{x}.\varphi, \delta, t) \rightarrowtail (\varphi, \delta[x \mapsto d], t)$ *for some* $d \in \mathbb{D}$*;*
- *if* $(\forall \boldsymbol{x}.\varphi, \delta, t) \in A$*, then* $(\varphi, \delta[x \mapsto d], t) \in A$ *and* $(\forall \boldsymbol{x}.\varphi, \delta, t) \rightarrowtail (\varphi, \delta[x \mapsto d], t)$ *for all* $d \in \mathbb{D}$*;*
- *if* $(\varphi \vee \psi, \delta, t) \in A$*, then* $(\varphi, \delta, t) \in A$ *and* $(\varphi \vee \psi, \delta, t) \rightarrowtail (\varphi, \delta, t)$*, or* $(\psi, \delta, t) \in A$ *and* $(\varphi \vee \psi, \delta, t) \rightarrowtail (\psi, \delta, t)$*;*
- *if* $(\varphi \wedge \psi, \delta, t) \in A$*, then* $(\varphi, \delta, t) \in A$*,* $(\psi, \delta, t) \in A$*,* $(\varphi \wedge \psi, \delta, t) \rightarrowtail (\varphi, \delta, t)$*, and* $(\varphi \wedge \psi, \delta, t) \rightarrowtail (\psi, \delta, t)$*;*
- *if* $(\mathsf{max}\, X.(\varphi_X), \delta, t) \in A$*, then* $(\varphi_X, \delta, t) \in A$ *and* $(\mathsf{max}\, X.(\varphi_X), \delta, t) \rightarrowtail (\varphi_X, \delta, t)$*;*

- *if* $(\mathsf{min}\,X.(\varphi_X),\delta,t)\in A$*, then* $(\varphi_X,\delta,t)\in A$ *and* $(\mathsf{min}\,X.(\varphi_X),\delta,t)\rightarrowtail(\varphi_X,\delta,t)$*; and*
- *if* $(X,\delta,t)\in A$*, then* $(\varphi_X,\delta,t)\in A$ *and* $(X,\delta,t)\rightarrowtail(\varphi_X,\delta,t)$.

Given an annotation $(A,\rightarrowtail)$ *and* $X\in FV\textsc{ar}$*, we define the relation* $\overset{X}{\rightarrowtail}\subseteq\rightarrowtail$ *thus:* $(\varphi,\delta,t)\overset{X}{\rightarrowtail}(\psi,\delta',u)$ *if and only if* $(\varphi,\delta,t)\rightarrowtail(\psi,\delta',u)$ *and* $\psi\neq Y$ *for any* $Y\in FV\textsc{ar}$ *such that* $X<Y$*. We say that* $(A,\rightarrowtail)$ *is* lfp-consistent *if there is no lfp variable* X *that appears infinitely often on a* $\overset{X}{\rightarrowtail}$*-path. For a formula* φ*, a data valuation* $\delta\in DE\textsc{nv}$ *and trace* t*, we say that* $(A,\rightarrowtail)$ *is an annotation for* φ,δ *on* t *(equivalently,* φ,δ *have annotation* $(A,\rightarrowtail)$ *on* t*) if*

1. $A\subseteq\mathtt{sub}(\varphi)\times DE\textsc{nv}\times\mathtt{suffix}(t)$;
2. $(\varphi,\delta,t)\in A$*; and*
3. $(A,\rightarrowtail)$ *is lfp-consistent.*

We say that $(A,\rightarrowtail)$ *is a* finite annotation *for* φ,δ *on* t *when* A *is finite and* $(A,\rightarrowtail)$ *is acyclic.*

Note that all conditions are local, except for lfp-consistency that allows us to distinguish between least and greatest fixed-points by forbidding least fixed-points to be unfolded infinitely often without encountering larger recursion variables.

▶ **Example 6.** Consider the formula φ_{leak} in Equation (4) on trace $u=0201^\omega$ starting from the empty data valuation. A minimal annotation witnessing that $u\in[\![\varphi_{\text{leak}}]\!]$ is:

$$\begin{aligned}
&(\exists\boldsymbol{x}.\langle x=\star\rangle\,\mathsf{min}\,X.((\langle x=\star\rangle\mathsf{tt}\vee\langle x\neq\star\rangle X)),\varnothing,0201^\omega)\\
&\rightarrowtail(\langle x=\star\rangle\,\mathsf{min}\,X.((\langle x=\star\rangle\mathsf{tt}\vee\langle x\neq\star\rangle X)),x\mapsto 0,0201^\omega)\\
&\rightarrowtail(\mathsf{min}\,X.((\langle x=\star\rangle\mathsf{tt}\vee\langle x\neq\star\rangle X)),x\mapsto 0,201^\omega)\\
&\rightarrowtail(\langle x=\star\rangle\mathsf{tt}\vee\langle x\neq\star\rangle X,x\mapsto 0,201^\omega)\\
&\rightarrowtail(\langle x\neq\star\rangle X,x\mapsto 0,201^\omega)\rightarrowtail(X,x\mapsto 0,01^\omega)\\
&\rightarrowtail(\mathsf{min}\,X.((\langle x=\star\rangle\mathsf{tt}\vee\langle x\neq\star\rangle X)),x\mapsto 0,01^\omega)\\
&\rightarrowtail(\langle x=\star\rangle\mathsf{tt}\vee\langle x\neq\star\rangle X,x\mapsto 0,01^\omega)\\
&\rightarrowtail(\langle x=\star\rangle\mathsf{tt},x\mapsto 0,01^\omega)\rightarrowtail(\mathsf{tt},x\mapsto 0,1^\omega)
\end{aligned}$$

The following result states that annotations do yield an alternative semantics for μHML^d:

▶ **Proposition 7.** *For every closed* $\varphi\in\mu HML^d$*, all* $\delta\in DE\textsc{nv}$ *and all* $t\in T\textsc{rc}$*:* φ,δ *have an annotation on* t *if and only if* $t\in[\![\varphi,\delta]\!]$.

Proof sketch. The left-to-right direction follows from the definition, except for fixed points that need to be inductively unfolded using Proposition 1. The lfp-consistency of the annotation allows us to use well-founded induction on the annotation. Conversely, if a trace satisfies a formula, one reconstructs an annotation using the iterative characterisation of fixed points [21, Section 3], in the same spirit as [1, Lemma 2.12 and Theorem 2.13], using transfinite induction to ensure that the constructed annotation is lfp-consistent. ◀

The annotations used in the proof of Proposition 7 may be infinite. However, the only rule that induces cycles or infinite unfolding is max, and only ∀ requires infinite branching (and indeed, the below proposition fails for them). Thus, closed formulae in cHML^d that have an annotation on some trace w always admit a finite one. Overall:

▶ **Proposition 8.** *Let* $\varphi\in cHML^d$*,* $\delta\in DE\textsc{nv}$ *and* $t\in T\textsc{rc}$*. Then,* $t\in[\![\varphi,\delta]\!]$ if and only if φ,δ *have a finite annotation on* t.

▶ **Corollary 9.** *The satisfiability problem for* cHML^d *is recursively enumerable.*

3 Complete and Satisfaction-Complete Fragments

3.1 Monitorability

The goal of this paper is to determine which properties can be verified at runtime. Informally, runtime verification is conducted as follows: along its execution, the *system under scrutiny* produces a trace, whose elements carry information about its operations; it can be thought of as a dynamically produced log file. We do not assume that the system terminates, so the trace is infinite, but termination can obviously be modelled e.g. by a termination symbol.

In parallel with the execution of the system, a program, called *monitor*, passively reads each element of the execution trace in an on-line manner. At any time, the monitor can emit a yes (respectively, a no) verdict, meaning that it considers that the system under scrutiny satisfies (resp., violates) a given specification. We then say that the monitor *accepts* (respectively, *rejects*) the trace. Note, however, that a monitor may never emit a verdict on reading an execution trace, e.g. if it does not have enough information to conclude. In this paper, we focus on *irrevocable* verdicts, meaning that once a verdict is emitted, the monitor cannot change its mind. Thus, for now, we can define a monitor through its acceptance and rejectance predicates (which will later on be defined through an operational model). Our definitions are inspired from [4, 6], although similar ideas already appeared earlier in the literature [50].

▶ **Definition 10** ([4, Definition 3.2 and Theorem 4.8], [6, Definitions 3.1 and 3.4])**.** *A monitor is an object m on which two predicates $\textbf{acc}(m,w)$ and $\textbf{rej}(m,w)$ are defined for all finite traces $w \in F\textsc{Trc}$, which satisfy the following properties:*

Consistency: *There is no finite trace w such that both $\textbf{acc}(m,w)$ and $\textbf{rej}(m,w)$ hold;*

Irrevocability: *For all $w,y \in F\textsc{Trc}$ such that $w \preceq y$, $\textbf{acc}(m,w) \Rightarrow \textbf{acc}(m,y)$ and $\textbf{rej}(m,w) \Rightarrow \textbf{rej}(m,y)$.*

We extend the definitions to infinite traces: for all $t \in \textsc{Trc}$, $\textbf{acc}(m,t)$ iff there exists some $w \prec t$ such that $\textbf{acc}(m,w)$, and similarly for $\textbf{rej}(m,t)$. Finally, a (finite or infinite) trace $u \in F\textsc{Trc} \cup \textsc{Trc}$ is accepted *(respectively,* rejected*) by m whenever $\textbf{acc}(m,u)$ (resp., $\textbf{rej}(m,u)$).*

Note that the irrevocability criterion implies that monitors only recognise suffix-closed languages, in the sense that both the sets of accepted and rejected traces of a monitor are suffix-closed. We can now relate monitors and properties:

▶ **Definition 11** ([4, Definition 4.1], [6, Definitions 3.3 and 3.5])**.** *Let $T \subseteq \textsc{Trc}$ be a property of traces, and m be a monitor.*

We say that m is sound *for satisfactions (respectively, for violations) for T if for all $t \in \textsc{Trc}$, $\textbf{acc}(m,t) \Rightarrow t \in T$ (respectively, $\textbf{rej}(m,t) \Rightarrow t \notin T$). We say that m is* sound *when it is sound for both satisfactions and violations.*

Conversely, we say that m is complete *for satisfactions (resp., violations) for T if the converse holds,* i.e., *for all $t \in \textsc{Trc}$, $t \in T \Rightarrow \textbf{acc}(m,t)$ (resp., $t \notin T \Rightarrow \textbf{rej}(m,t)$). We then say that T is* completely monitorable for satisfactions *(resp.,* for violations*). We say that m is* complete *if it is complete for both, and correspondingly that T is* completely monitorable.

We say that the above are effective *when m can be computed by a Turing machine.*

We extend those definitions to any formula $\varphi \in \mu HML^d$ by considering $T = [\![\varphi]\!]$.

In plain words, a property is completely monitorable if there exists a monitor that detects all its satisfactions and violations. Monitorability is thus defined relative to a monitor model, since it depends on the computational power of the monitoring program. As a first step, we consider monitors with arbitrary power; we do not even assume that they are computable.

This very strong definition is to be thought of as an overapproximation. However, as witnessed by Theorem 14, a quite weak monitor model suffices, since completely monitorable properties turn out to be very simple. This motivates the study of satisfaction-completeness in Secs. 3.3 and 4, as well as that of optimal monitors (Definition 13 below) in Sec. 3.4.

For now, a monitor is to be conceived simply as a machine (possibly with access to arbitrary oracles) m which processes a trace and possibly eventually raises a yes or a no verdict. When monitoring for some formula $\varphi \in \mu\mathrm{HML}^d$, if m emits yes upon reading a finite trace $w \in \mathrm{FTRC}$, it means that any continuation $wt \in \mathbb{D}^\omega$ belongs to $[\![\varphi]\!]$. Conversely, if it emits a no, it means that $w\mathbb{D}^\omega \cap [\![\varphi]\!] = \varnothing$. Thus, as long as we are not concerned with the way it executes, a monitor m is fully described by the set of prefixes for which it emits a verdict. Observe that T is completely monitorable iff there exist two sets $G, B \subseteq \mathrm{FTRC}$ such that $T = G\mathbb{D}^\omega = \mathrm{TRC} \setminus (B\mathbb{D}^\omega)$, *i.e.,* it is characterised by its good and bad prefixes.

▶ **Definition 12** ([9, 26]). *Let $T \subseteq \mathit{TRC}$ be a set of traces. We say that $w \in \mathit{FTRC}$ is a* good *(respectively, a* bad*) prefix for T when $w\mathbb{D}^\omega \subseteq T$ (respectively, $w\mathbb{D}^\omega \cap T = \varnothing$).*

▶ **Definition 13** ([5, Definition 10]). *Let $T \subseteq \mathit{TRC}$ be a property of traces and MON be a set of monitors. A monitor $m \in \mathit{MON}$ is* optimal for violations *(respectively, for satisfactions) in MON for T if for each monitor $m' \in \mathit{MON}$ that is sound for violations (resp., for satisfactions) for T and each $t \in \mathit{TRC}$, if $\boldsymbol{rej}(m', t)$ then $\boldsymbol{rej}(m, t)$ (resp., if $\boldsymbol{acc}(m', t)$ then $\boldsymbol{acc}(m, t)$).*

If one considers arbitrary monitors (including non-computable ones), we can then say that a monitor m is violation-optimal *for T if for all $w \in \mathit{FTRC}$, if w is a bad prefix for T then $\boldsymbol{rej}(m, w)$, and dually for satisfaction-optimality.*

3.2 The Complete Fragment: HML^d

In the finite alphabet case, all completely monitorable formulae can be expressed in the fragment HML, which consists in formulae of μHML without recursion [4, Theorem 4.8]. The proof of that result can be adapted to establish the following theorem, taming the infinity of the domain by quotienting finite traces by bijections over $\mathbb{D}$ (HML^d is defined in Fig. 1).

▶ **Theorem 14.** *Let $T \subseteq \mathit{TRC}$ be a set of traces that is stable under renamings (*i.e.*, for all bijections $\sigma : \mathbb{D} \to \mathbb{D}$, $\sigma(T) = T$). T is completely monitorable* iff *it can be expressed in HML^d.*

▶ Remark 15. This result does not hold if we consider the domain $(\mathbb{N}, <)$. Indeed, there, one can define the set $D = \{d_0 d_1 \ldots d_n \# w \mid \forall i < j, d_i > d_j\}$, which is completely monitorable, since n is bounded by d_0, but cannot be expressed in HML^d since n depends on d_0.

3.3 A Satisfaction-Complete Fragment: cHML^d

Having to detect *all* satisfactions and *all* violations prevents us from monitoring for behaviours that can happen after an unbounded number of steps in a system execution, restricting us to the tiny fragment HML^d, where properties cannot be recursive. In the following, we relax our notion of completeness and focus on detecting only one kind of verdict, *i.e.,* single-verdict monitors. We also consider the richer setting of optimal monitors in Sec. 3.4, but most results are unfortunately negative. Without loss of generality, we consider satisfaction-completeness, the ability to detect all satisfactions of a property. In practice, as reflected in the literature, runtime verification is more focussed on detecting violations, which are often more critical. Since this adds one level of negation and hence of technicality, we work with satisfactions and results about violation-completeness are obtained by duality.

Syntax. $m, n \in \textsc{Mon} ::= \mathsf{yes} \mid \mathsf{end} \mid (b).m \mid \mathsf{guess}\ x.m \mid m \oplus n \mid m \otimes n \mid \mathsf{rec}\, X.m \mid X$

Configurations. $c \in C ::= (m, \delta) \mid c \odot c$, where $m \in \textsc{Mon}$ is a monitor, $\delta \in \mathrm{D}\textsc{Env}$ is a data environment and $\odot$ is either $\oplus$ (parallel OR) or $\otimes$ (parallel AND).

Small-Step Semantics.

$$\textsc{mVrd}\ \frac{v \in \{\mathsf{yes}, \mathsf{end}\}}{v, \delta \xrightarrow{d} v, \delta} \qquad \textsc{mAct}\ \frac{b\delta[\star \mapsto d] \Downarrow \mathsf{true}}{(b).m, \delta \xrightarrow{d} m, \delta}\ d \in \mathbb{D} \qquad \textsc{mBlc}\ \frac{b\delta[\star \mapsto d] \Downarrow \mathsf{false}}{(b).m, \delta \xrightarrow{d} \mathsf{end}, \delta}\ d \in \mathbb{D}$$

$$\textsc{mGs}\ \frac{}{\mathsf{guess}\ x.m, \delta \xrightarrow{\tau} m, \delta[x \mapsto d]}\ d \in \mathbb{D} \qquad \textsc{mRec}\ \frac{}{\mathsf{rec}\, X.m, \delta \xrightarrow{\tau} m[{}^{\mathsf{rec}\, X.m}/_X], \delta}$$

$$\textsc{mFork}\ \frac{}{m \odot n, \delta \xrightarrow{\tau} m, \delta \odot n, \delta} \qquad \textsc{mSyn}\ \frac{c_1 \xrightarrow{d} c_1' \qquad c_2 \xrightarrow{d} c_2'}{c_1 \odot c_2 \xrightarrow{d} c_1' \odot c_2'}$$

$$\textsc{mAsync}_{\mathrm{L}}\ \frac{c_1 \xrightarrow{\tau} c_1'}{c_1 \odot c_2 \xrightarrow{\tau} c_1' \odot c_2} \qquad \textsc{mAsync}_{\mathrm{R}}\ \frac{c_2 \xrightarrow{\tau} c_2'}{c_1 \odot c_2 \xrightarrow{\tau} c_1 \odot c_2'}$$

$$\textsc{mVrC1}\ \frac{}{\mathsf{yes}, \delta \otimes c \xrightarrow{\tau} c} \qquad \textsc{mVrD2}\ \frac{}{\mathsf{yes}, \delta \oplus c \xrightarrow{\tau} \mathsf{yes}, \delta}$$

Synthesis.

$$(\!|\mathsf{tt}|\!) = \mathsf{yes} \qquad (\!|\exists \boldsymbol{x}.\varphi|\!) = \mathsf{guess}\ x.(\!|\varphi|\!) \qquad (\!|\langle b \rangle \varphi|\!) = (b).(\!|\varphi|\!)$$

$$(\!|\varphi \vee \psi|\!) = (\!|\varphi|\!) \oplus (\!|\psi|\!) \qquad (\!|\varphi \wedge \psi|\!) = (\!|\varphi|\!) \otimes (\!|\psi|\!) \qquad (\!|\mathsf{min}\, X.(\varphi)|\!) = \mathsf{rec}\, X.(\!|\varphi|\!) \qquad (\!|X|\!) = X$$

Figure 2 Syntax, small-step semantics and synthesis of monitors.

Monitor Synthesis. In Figure 2, we introduce a model of monitors, along with a synthesis procedure. We now show that it yields sound and satisfaction-complete monitors for formulae in cHML^d (defined in Fig. 1 on page 5). Note that this fragment includes conjunctions, and it can express $\mathsf{ff} \equiv \langle \bot \rangle \mathsf{tt}$ (where $\bot$ stands e.g. for $x \neq x$) and (linear-time) necessity.

To keep track of the value of each data variable, a monitor $m \in \textsc{Mon}$ is equipped with a data environment $\delta \in \mathrm{D}\textsc{Env}$ forming a pair (m, δ). It begins its execution in the context of an initial data environment δ_0, as a single component (m, δ_0). Unless otherwise stated, $\delta_0 = \varnothing$. Note that for closed monitors, the semantics do not depend on δ. Along its execution, a monitor might fork into parallel components. On forking, each component receives a local copy of the parent monitor's data environment (rule mFork) and then evolves independently. The only way to recombine components is when (at least) one has raised a verdict. The verdict is then aggregated with the other components following the usual rules of propositional logic, where yes corresponds to $\top$, $\oplus$ to $\vee$ and $\otimes$ to $\wedge$ (rules mVr). To simulate existential quantification, a monitor can non-deterministically guess the value of a data variable and store it in its data environment (rule mGs). This overwrites a previous valuation if any.

There are two kinds of transitions. Ones of the form $c \xrightarrow{\tau} c'$ are called τ-transitions, and correspond to internal moves of the monitor, that happen without reading any trace elements. Correspondingly, τ is such that for all (finite or infinite) traces $u \in \mathrm{F}\textsc{Trc} \cup \textsc{Trc}$, $\tau u = u$. Those of the form $c \xrightarrow{d} c'$, for $d \in \mathbb{D}$, are transitions that *process* an element from the trace.

For two configurations c,c' and a data value d, we write $c \overset{d}{\Longrightarrow} c'$ whenever $c \overset{\tau}{\rightarrow}^{*} c'' \overset{d}{\rightarrow} c''' \overset{\tau}{\rightarrow}^{*} c'$ for some configurations c'' and c'''. For a finite trace $w = d_0 d_1 \dots d_l$, we then write $c \overset{w}{\Longrightarrow} c'$ whenever $c \overset{d_0}{\Longrightarrow} c_1 \overset{d_1}{\Longrightarrow} c_2 \dots c_l \overset{d_l}{\Longrightarrow} c'$. By a slight abuse of notation, for all $t \in \textsc{Trc}$, we define $\mathbf{acc}(c,t)$ as $c \overset{w}{\Longrightarrow} \mathsf{yes}, \delta'$ for some $\delta' \in \textsc{DEnv}$ and some $w \prec t$.

▶ **Example 16.** Consider a server that issues identifier tokens. Assume that the first token it issues is its own and should not be leaked, *i.e.,* that the server *does not* satisfy the formula $\varphi_{\text{leak}} = \exists \boldsymbol{x}. \langle x = \star \rangle \mathsf{min}\, X. (\langle x = \star \rangle \mathsf{tt} \vee \langle x \neq \star \rangle X)$ (already encountered in Ex. 2). The procedure of Fig. 2 yields $m_{\text{leak}} \triangleq (\!|\varphi_{\text{leak}}|\!) = \mathsf{guess}\ x.(\star = x).\mathsf{rec}\, X.((\star = x).\mathsf{yes} \oplus (\star \neq x).X)$.

Consider an erroneous execution "1.0.1.... " exhibited by the server. m_{leak} starts in configuration $\mathsf{guess}\ x.(\star = x).\mathsf{rec}\, X.((\star = x).\mathsf{yes} \oplus (\star \neq x).X), \varnothing$. Following rule mGs, m_{leak} *internally* selects a concrete value $d \in \mathbb{D}$ for x. Note that such a value is selected over a possibly infinite domain, reminiscent of [10]. Assume it chooses the value 0 for x. On the next step, the system emits 1 and the monitor checks for the guard $(\star = y)$, which does not hold. Following rule mBlc, it transitions to the inconclusive verdict $\mathsf{end}, x \mapsto 1$, where it stays forever. Assume instead that the monitor picks $x = 1$. Then, we have: $m_{\text{leak}}, \varnothing \xrightarrow[\text{mGs}]{\tau} (\star = x).\mathsf{rec}\, X.((\star = x).\mathsf{yes} \oplus (\star \neq x).X), x \mapsto 1$.

The execution of the monitor continues and it eventually raises a yes verdict. Thus, the trace is accepted by the monitor: it recognises that the system repeats its first action, and hence violates its specification. Note the importance of the non-deterministic choice of a value for x using rule mGs. ⌟

It would not be difficult to establish that m_{leak} is a sound and satisfaction-complete monitor for φ_{leak}. This is more generally the case for cHML^d, and dually for sHML^d:

▶ **Theorem 17.** *cHML^d (respectively, sHML^d) is completely monitorable for satisfactions (resp., violations).*

Proof sketch. Soundness of the synthesis procedure of Fig. 2 is proven similarly to [4, Prop. 4.15]. There, the proof is written for violations but is easily adapted, and data variables do not interfere: they play the same role in monitors as in formulae.

To prove satisfaction-completeness, we use the annotation semantics of Sec. 2.4: in essence, monitors compute annotations of cHML^d formulae, so from an annotation of $\varphi \in$ cHML^d, one can build an accepting run of $(\!|\varphi|\!)$. ◀

Monitors and Register Automata. We conclude by observing that our model of monitors for cHML^d is equivalent to a model of register automata. This result echoes the equivalence between alternation-free modal μ-calculus and register tree automata in [46, Theorems 3 and 7]. With the same ingredients, one can adapt the one in [3, Section 4.2].

Register automata were introduced in [45] as *finite-memory automata.* They consist in a finite-state automaton equipped with a finite set of *registers*, that can store values from an infinite domain (here, $\mathbb{D}$). It is able to compare the value it reads with the content of its registers, and transition accordingly. For a formal definition, see [18, Section 1.3]; one can omit labels which play no role here.

▶ **Theorem 18.** *Let $L \subseteq \mathbb{D}^*$ be a suffix-closed language. There exists an alternating register automaton with existential guessing that recognises L* if and only if *there exists a monitor that accepts exactly the traces in L.*

The correspondence also holds between register automata with no universal (respectively, existential) states and monitors with no $\otimes$ (resp., $\oplus$). Moreover, if one defines $\mathsf{match}(r,b) \triangleq \mathsf{guess}\ r.(b \wedge r = \star)$, all the above correspondences hold for register automata with no guessing and monitors whose $\mathsf{guess}\ r$ construct is replaced with the $\mathsf{match}(r,b)$ one. This implies that a similar equivalence holds between the monitors in [2] and register automata without non-deterministic guessing.

Since all those classes of register automata are inequivalent [18, Section 1.5], we know that all those variants of monitors correspond to different classes of properties. Thus, in cHML^d and sHML^d, removing conjunctions or disjunctions reduces expressiveness, and the same holds when replacing existential quantification with a $\mathsf{match}(r,b)$ construct (as defined in [7]). This also shows that deterministic monitors (defined as the counterpart of deterministic register automata) are strictly less expressive than non-deterministic or alternating ones, which invalidates [7, Theorem 18].

3.4 Optimal Monitors

The main obstacle to complete monitorability is that of behaviours that happen in the limit, which obviously cannot be monitored for at runtime. For instance, no monitor can ever detect that there are only finitely many occurrences of a given data value. In the spirit of [5], we thus consider *optimal monitors*, that are only required to flag all violations or satisfactions that may be detected by some monitor.

First, disjHML^d (as defined in Fig. 1) is equivalent to non-deterministic register automata. Their emptiness problem is decidable [45, Theorem 1], so we can build optimal monitors:

▶ **Theorem 19.** *For all $\varphi \in$ disjHML^d, one can effectively construct a monitor that is satisfaction-complete and violation-optimal for φ.*

Yet, as soon as we add conjunctions to get cHML^d, this becomes impossible. Indeed, from a violation-optimal monitor one can build a semi-algorithm to decide unsatisfiability of cHML^d. Since we have a semi-algorithm for satisfiability of cHML^d (Corollary 9), this would yield an algorithm to decide satisfiability of cHML^d, contradicting Theorem 4.

▶ **Theorem 20.** *No effective procedure can construct violation-optimal monitors for cHML^d.*

4 Satisfaction-Completeness: Beyond cHML^d

We just established that cHML^d is a fragment of μHML^d that can be monitored in a sound and satisfaction-complete way with the synthesis procedure of Fig. 2 (Theorem 17), which generalises the finite alphabet case [4, Proposition 4.15]. Moreover, our model of monitors is expressively equivalent to register automata (Theorem 18), which generalises [3, Section 4.2], with the major difference that our monitors cannot be made deterministic.

We now show that, in contrast to the finite alphabet case [4, Proposition 4.18], that fragment is however not maximal: there are properties that admit sound and satisfaction-complete monitors that cannot be expressed in cHML^d. Proposition 21 presents such a property, which can be expressed in the larger fragment minHML$^d_{\forall_g}$ that we introduce. The latter is “almost maximal”: it is maximal *within* minHML^d, *i.e.,* μHML^d without greatest fixed-points (Theorem 26), but not in the full μHML^d. This is for a good reason: there cannot exist a maximally monitorable fragment of μHML^d that is effective (Corollary 29).

4.1 A Candidate Maximal Fragment...

In general, universally quantified formulae are not monitorable for satifactions, as they require checking infinitely many instantiations of the quantified variable. Consider, e.g., the formula $\forall \boldsymbol{x}.\mathsf{min}\, X.(\langle\star = x\rangle \mathsf{tt} \vee \langle\star \neq x\rangle X)$ which states that all data values appear in the input. It is satisfiable, since we assumed that the data domain is countable. Yet, it is not monitorable for satisfactions: any finite prefix only contains finitely many data values and can be continued by, e.g., $\#^\omega$, yielding an input which violates the formula.

Nevertheless, some formulae containing universal quantifiers *are* monitorable. Consider the property which states that the input is divided into blocks separated by dollar and sharp symbols, and that all data values that appear in the second block appear in the first block (formalised in Equation 8). It is monitorable for satisfactions: the monitor reads the first two blocks by waiting to see the \$ and then the #; if this never happens it means that the input violates the property. Otherwise, the monitor can check that all data values in the second block appear in the first one by processing them one by one, going back and forth.

$$L_{\forall\#\exists\$} = \{d_1 \ldots d_k \$ e_1 \ldots e_l \# \ldots \mid \forall 1 \leq j \leq l, \exists 1 \leq i \leq k, d_i = e_j\}, \text{ expressed as:} \tag{8}$$
$$\varphi_{\forall\#\exists\$} = \exists \boldsymbol{x}.\gamma(x) \wedge \forall \boldsymbol{x}.(\gamma(x) \vee \psi(x)), \text{ where:}$$
$$\gamma(x) = \mathsf{min}\, X.(\langle\star \neq \$\rangle X \vee \langle\star = \$\rangle \mathsf{min}\, Y.(\langle\star = \#\rangle \mathsf{tt} \vee \langle\star \neq x\rangle Y))$$
$$\psi(x) = \mathsf{min}\, Z.(\langle\star = x\rangle \mathsf{tt} \vee \langle\star \neq \$\rangle Z)$$

The formula $\gamma(x)$ is called the *guard*, and sets a monitorable bound on the maximal position where a candidate x violating the formula ψ can be found. This way, once the bound is found, the monitor knows that the subsequent data values that appear need not be checked. Here, it expresses that the trace starts with two blocks – ended by \$ and #, respectively – and that x does not appear in the second block: first, look for a \$; once it is found, look for a #, and if in the meantime x is encountered, the formula cannot recurse and therefore rejects.

The formula $\psi(x)$ is the universally quantified property, and since we are looking for satisfactions, its universal quantification has to be limited to finitely many values to ensure that it has a finite witness, hence the disjunction with the guard. Here, it expresses that x appears in the first block. Summing up, the conjunct $\exists \boldsymbol{x}.\gamma(x)$ ensures that a trace has the form $w_1 \$ w_2 \# u$ for some $w_1, w_2 \in \mathbb{D}^*$ and $u \in \textsc{Trc}$, while the conjunct $\forall \boldsymbol{x}.(\gamma(x) \vee \psi(x))$ yields that every $d \in \mathbb{D}$ occurring in w_2 also occurs in w_1.

The above property cannot however be expressed in $\textsc{cHML}^d$. Indeed, the length of w_2 is unbounded, and its elements have to be compared to elements that appear *before* in the input, so they cannot be manipulated only using existential quantifiers, even with fixed points. This is made formal by going through monitors: by Theorem 17, the synthesis procedure of Fig. 2 yields sound a satisfaction-complete monitors. Now, those monitors can only carry boundedly many data values accross the \$ sign. Thus, $\textsc{cHML}^d$ is not a maximally monitorable fragment.

▶ **Proposition 21.** *There does not exist any formula $\varphi \in \textsc{cHML}^d$ such that $[\![\varphi]\!] = L_{\forall\#\exists\$}$.*

We now proceed to characterise the collection of formulae without greatest fixed points that can be monitored in a sound and satisfaction-complete fashion. Given a formula γ, a data variable x, and a finite set $F \subset \textsc{DVar}$ of data variables, we use the following notations:

$$Fx \triangleq F \cup \{x\}; \quad F\bar{x} \triangleq F \setminus \{x\}; \quad x \neq F \triangleq \bigwedge_{y \in F\bar{x}} \langle x \neq y\rangle \mathsf{tt}; \quad x \sim F \triangleq \bigvee_{y \in F\bar{x}} \langle x = y\rangle \mathsf{tt}; \quad \text{and}$$
$$\forall \boldsymbol{x} \leq \boldsymbol{\gamma} + \boldsymbol{F}.\ \varphi \triangleq \exists \boldsymbol{x}.(x \neq F \wedge \gamma) \wedge \forall \boldsymbol{x}.((x \neq F \wedge \gamma) \vee \varphi).$$

The formula $x \neq F$ describes that the value of x is different from every value assigned to any element of F (except for x if $x \in F$), and $x \sim F$ conversely describes that the value of x coincides with the value assigned to some other element of F.

The quantifier in $\forall \boldsymbol{x} \leq \boldsymbol{\gamma} + \boldsymbol{F}.\ \varphi$ intuitively bounds the quantification of x, as we only need to verify φ for all data values that are assigned to variables in $F\bar{x}$ and the ones for which γ is not true. As such, we say that γ is a guard or bound for x, or that x is guarded. We need to keep track of the free and guarded variables. Hence, we parameterize the definition of our fragment with respect to two finite sets of data variables. For all finite $F \subset \text{DVAR}$ and $V \subseteq F$, we define $\text{MINHML}^d_{\forall_g V,F}$ as the set of formulae that are produced from $\varphi_{V,F}$ in the following grammar whose grammar variables are parameterized with respect to V and F:

$$\varphi_{V,F}, \gamma_{V,F} ::= \mathsf{tt} \mid \mathsf{ff} \mid X \mid \min X.(\varphi_{V,F}) \mid \langle b(F) \wedge \star \neq V \rangle \varphi_{V,F} \mid \varphi_{V,F} \wedge \varphi_{V,F} \mid \varphi_{V,F} \vee \varphi_{V,F}$$
$$\mid \forall \boldsymbol{x} \leq \boldsymbol{\gamma_{Vx,Fx}} + \boldsymbol{F}.\varphi_{V\bar{x},Fx} \mid \exists \boldsymbol{x}.(x \neq V \wedge \varphi_{V\bar{x},Fx}) \vee (x \sim V \wedge \varphi_{Vx,Fx})$$

We then define $\text{MINHML}^d_{\forall_g} = \bigcup_{V \subseteq F \subset \text{DVAR}} \text{MINHML}^d_{\forall_g V,F}$. If $\varphi \in \text{MINHML}^d_{\forall_g}$ has no free data variables, then $\varphi \in \text{MINHML}^d_{\forall_g \varnothing,\varnothing}$. In the above grammar, the set F keeps track of the *free variables* in φ, and V of the "*guarded*" *free variables.* Here, "x is guarded" means that the value of x is not encountered in the trace while we evaluate the formula (but this value is still assigned to some variable). This is ensured by the "$\star \neq V$" conjunct in the diamonds and by guaranteeing that, during the existential quantification of y, if the value of y matches that of some x in V, then y is added to V. Hence $\varphi_{Vx,F}(x)$ can only be true for values of x that do not appear in some finite annotation of φ.

The main characteristic of this fragment is that every universal quantification is *guarded* by a bound on the positions where a candidate x violating the formula can be found. This is achieved by partitioning the potential values of x into those that appear during the (finite) evaluation of the guard (and must be checked against φ) and those that do not (and therefore satisfy the guard). Thus, when monitoring or evaluating $\forall \boldsymbol{x} \leq \boldsymbol{\gamma_{Vx,Fx}} + \boldsymbol{F}.\varphi_{V\bar{x},Fx}$, we only need to consider a fixed number of cases for the value of x when checking the subformula $\gamma_{Vx,Fx}$, and therefore, $\gamma_{Vx,Fx}$ is, in a sense, easier to monitor for, or evaluate, than $\varphi_{V\bar{x},Fx}$. Then, the number of cases that we need to consider for the value of x when checking the subformula $\varphi_{V\bar{x},Fx}$ is finite and depends on how we evaluated $\gamma_{Vx,Fx}$.

In $\varphi_{\forall\#\exists\$}$, the evaluation of the guard γ is complete at the end of the second block. Therefore, to evaluate $\forall x.(\gamma(x) \vee \psi(x))$, it suffices to check values of x for ψ that appear during the evaluation of γ – specifically in the second block. More generally, the grammar of $\text{MINHML}^d_{\forall_g}$ induces a recursive strategy to evaluate a formula while only remembering finitely many cases for the values assigned to its variables. As we see below, this allows us to find a finite witness for the satisfaction of a formula, using a guarded version of annotations, and, subsequently, to monitor for the satisfaction of all formulae in $\text{MINHML}^d_{\forall_g}$.

Guarded-branching Annotations. We can extend the definitions for annotations for CHML^d from Sec. 2.4 to guarded-branching annotations for MINHML^d. For annotation $(A, \rightarrowtail)$, we replace the quantifier conditions with:

if $\boldsymbol{a = (\forall x \leq \gamma + F.\varphi, \delta, t) \in A}$**,** there is some finite $D \cup \{d_*\} \subseteq \mathbb{D}$ such that $d_* \notin D$, and:

1. $(\gamma, \delta[x \mapsto d_*], t) \in A$ and $a \rightarrowtail (x \neq F \wedge \gamma, \delta[x \mapsto d_*], t)$;
2. for every $d \in D$, $(\varphi, \delta[x \mapsto d], t) \in A$ and $a \rightarrowtail (\varphi, \delta[x \mapsto d], t)$, or $(\gamma, \delta[x \mapsto d], t) \in A$ and $a \rightarrowtail (\gamma, \delta[x \mapsto d], t)$;
3. for every $d \in \mathbb{D}$, having transition $(\gamma, \delta[x \mapsto d_*], t) \rightarrowtail^* (\psi, \delta', du)$ implies $d \in D$; and
4. $\{\delta(x) \mid x \in \text{DVAR}\} \cup (F \cap \mathbb{D}) \subseteq D$.

if $\boldsymbol{a = (\exists x.(x \neq V \wedge \varphi_1) \vee (x \sim V \wedge \varphi_2),\delta,t),}$ then there is some $d \in \mathbb{D}$, such that either

- $d \neq \delta(y)$ for every $y \in V\bar{x}$, and $(\varphi_1,\delta[x \mapsto d],t) \in A$ and $a \rightarrowtail (\varphi_1,\delta[x \mapsto d],t)$; or
- $d = \delta(y)$ for some $y \in V\bar{x}$, and $(\varphi_2,\delta[x \mapsto d],t) \in A$ and $a \rightarrowtail (\varphi_2,\delta[x \mapsto d],t)$.

The condition for the existential quantifier is used to delineate the existential quantifier as it appears in the grammar and the one hidden inside the guarded universal quantifier.

▶ **Theorem 22.** *For every closed $\varphi \in$ MINHML$^d_{\forall_g}$, $\delta \in$ DENV, and $t \in$ TRC, $t \in [\![\varphi,\delta]\!]$* if and only if *$(\varphi,\delta,t)$ has a finite guarded-branching annotation.*

Proof Sketch. For guarded variables (the ones in V) and for variables whose value does not appear in the annotation, the specific value does not affect the evaluation of the formula, which allows us to show the equivalence of annotations with (finite) guarded-branching ones. For the universal quantifier, D represents the values that we must explicitly consider and $d_\star$ is a "dummy" value that represents all other values. ◀

The Monitorable Least-fixed-point formulae. The guarded-branching annotation semantics for MINHML$^d_{\forall_g}$ yields that the fragment is effectively monitorable for satisfactions, in the sense that satisfactions can be monitored by a Turing machine. The monitors of Sec. 3.3 are equivalent to alternating register automata (Theorem 18), which are computable, so:

▶ **Theorem 23.** *Every formula in* CHML^d *is effectively monitorable for satisfactions.*

Now, as a consequence of Theorem 22:

▶ **Corollary 24.** *Let $\varphi \in$ MINHML$^d_{\forall_g}$. If $t \in [\![\varphi]\!]$, then t has a good prefix for φ.*

▶ **Corollary 25.** *Every $\varphi \in$ MINHML$^d_{\forall_g}$ is effectively monitorable for satisfactions.*

Moreover, the fragment MINHML$^d_{\forall_g}$ *characterizes* the monitorable properties in MINHML^d. There, not all formulae are monitorable, but they are optimally effectively monitorable for satisfactions, in the sense that there exists a satisfaction-optimal monitor for them:

▶ **Theorem 26.** *Every formula $\varphi \in$ MINHML^d is optimally effectively monitorable for satisfactions. A formula $\varphi \in$ MINHML^d is monitorable if and only if $\varphi \equiv \psi$ (*i.e., $[\![\varphi]\!] = [\![\psi]\!]$*) for some $\psi \in$ MINHML$^d_{\forall_g}$.*

To prove this theorem, we introduce gd, that turns every MINHML^d formula into a guarded form in MINHML$^d_{\forall_g}$. Let φ be a closed formula without max operators and let $Vr(\varphi) \subseteq$ DVAR be the set of data variables that appear in φ. For every subformula ψ of φ, finite $V \subseteq F \subset Vr(\varphi)$, let $X_{V,F}$ be a new recursion variable associated with X and V,F. For each finite $\Pi \subseteq (2^{Vr(\varphi)})^2$, we define $\mathsf{gd}(\psi,V,F,\Pi)$ by double recursion on $(2^{Vr(\varphi)})^2 \setminus \Pi$ and ψ:

- $\mathsf{gd}(\psi,V,F,\Pi) = \psi$, when $\psi = \mathtt{tt}$ or $\psi = \mathtt{ff}$;
- $\mathsf{gd}(X,V,F,\Pi) = X_{V,F}$, when $(V,F) \in \Pi$;
- $\mathsf{gd}(X,V,F,\Pi) = \mathsf{gd}(\mathtt{fx}(X),V,F,\Pi)$, when $(V,F) \notin \Pi$;
- $\mathsf{gd}(\mathsf{min}X.\psi,V,F,\Pi) = \mathsf{min}X_{V,F}.\mathsf{gd}(\psi,V,F,\Pi \cup \{(V,F)\})$;
- $\mathsf{gd}(\forall x.\psi,V,F,\Pi) = \forall \boldsymbol{x \leq \mathsf{gd}(\psi, Vx, Fx, \Pi) + F}.\mathsf{gd}(\psi,V\bar{x},Fx,\Pi)$;
- $\mathsf{gd}(\exists x.\psi,V,F,\Pi) = \exists x.(x \neq V \wedge \mathsf{gd}(\psi,V\bar{x},Fx,\Pi)) \vee (x \sim V \wedge \mathsf{gd}(\psi,Vx,Fx,\Pi))$;
- $\mathsf{gd}(\langle b\rangle\psi,V,F,\Pi) = \langle b \wedge \bigwedge_{x\in V}(x \neq *)\rangle \mathsf{gd}(\psi,V,F,\Pi)$;

and $\mathtt{gd}(-,V,F,\Pi)$ commutes with $\wedge$ and $\vee$. Observe that for all ψ and $V \subseteq Var$, $\mathtt{gd}(\psi,V,F,\Pi) \in \textsc{min}\mathrm{HML}^d_{\forall_g V,F}$. We then define $\mathtt{gd}(\psi,V,F) = \mathtt{gd}(\psi,V,F,\varnothing)$ and $\mathtt{gd}(\psi) = \mathtt{gd}(\psi,\varnothing,\varnothing)$, where ψ has no free recursion variables, and, respectively, no free data variables.

The idea behind $\mathtt{gd}$ is to leverage the existence of *good prefixes* for a formula to construct a formula in the guarded fragment. To do so, $\mathtt{gd}$ guards the universal quantification in $\forall x.\psi(x)$ by a "*more monitorable*" formula $\gamma(x)$ that is constructed from $\psi(x)$ by guarding x. Intuitively, a *good prefix* p for $\gamma(x)$ (which exists if the trace is a satisfying one and the formula/guard is monitorable) provides a bound on the part of the trace to consider when looking for candidate values violating $\psi(x)$. Data values outside p are irrelevant: they satisfy $\gamma(x)$ and do not need to be verified against $\psi(x)$.

The operation $\mathtt{gd}$ produces formulae with good monitorability properties when applied to formulae in $\textsc{min}\mathrm{HML}^d$. In fact, for each $\varphi \in \textsc{min}\mathrm{HML}^d$, $\mathtt{gd}(\varphi) \in \textsc{min}\mathrm{HML}^d_{\forall_g}$ and therefore is monitorable for satisfactions. Furthermore, the sound and complete monitor for $\mathtt{gd}(\varphi)$ is *optimal* for φ, in that it can detect all good prefixes for φ; finally, if φ is monitorable for satisfactions, then φ and $\mathtt{gd}(\varphi)$ are equivalent.

4.2 ... That Is Not Maximal in General

In the finite alphabet case, one can turn greatest fixed points into least fixed points while preserving monitorable consequences by a procedure analogous to determinisation of word automata [5, Section 5]. Over data domains, this is not the case anymore [45, Section 4]. In this section, we show that the addition of $\mathtt{max}$ strictly increases the monitorable fragment, and establish that it is undecidable to check if a formula is (effectively) monitorable.

▶ **Lemma 27.** *For each deterministic Turing machine M, we can construct a formula:*

1. $\psi^e_M \in s\mathrm{HML}^d$*, such that* $[\![\psi^e_M]\!]$ *is the set of traces that encode the run of M on 0; and*
2. $\psi^{\neg H}_M$*, such that* $[\![\psi^{\neg H}_M]\!]$ *is the set of traces that encode a non-empty prefix of a run of M, but do not encode a* terminating *run of M.*

▶ **Corollary 28.** $\psi^{\neg H}_T \in \mu\mathrm{HML}^d$ *is monitorable for satisfactions, but not effectively monitorable for satisfactions for every T.*

Proof. The formula $\psi^{\neg H}_T \in \mu\mathrm{HML}^d$ is monitorable for satisfactions, because every satisfying trace t extends a finite trace p that encodes the starting configuration of T on input x. Indeed, if T on x terminates, then every satisfying trace is not a correct encoding of a run of T, and therefore has a good prefix. If T on x does not terminate, then every extension of p satisfies the formula, and therefore p is a good prefix. Therefore, every satisfying t has a good prefix that extends p, yielding that $\psi^{\neg H}_T \in \mu\mathrm{HML}^d$ is monitorable for satisfactions.

If $\psi^{\neg H}_T$ were effectively monitorable, then there would exist a Turing machine M that would recognize the good prefixes of $\psi^{\neg H}_T$. M would accept x whenever T does not terminate on x, yielding that the Halting problem is co-recursively enumerable, which is a contradiction. ◀

▶ **Corollary 29.** *Monitorability and effective monitorability for satisfactions for* $s\mathrm{HML}^d$ *and* $\mu\mathrm{HML}^d$ *are undecidable.*

Proof. Observe that ψ^e_T is (effectively) monitorable for satisfactions if and only if T terminates on 0: if T on 0 terminates, then every satisfying trace has a good prefix where an error in the encoding has occurred, or where the full encoding of a run has appeared. Conversely, if T on 0 does not terminate, then the trace that encodes the run of T on 0 has no good prefix, as every prefix can be extended in a way that does not encode the run. ◀

The above result yields the impossibility of a decidable, maximal monitorable fragment of $\mu\mathrm{HML}^d$, and similarly for an effectively monitorable fragment.

References

1 Luca Aceto, Antonis Achilleos, Elli Anastasiadi, Adrian Francalanza, and Anna Ingólfsdóttir. Complexity results for modal logic with recursion via translations and tableaux. *Logical Methods in Computer Science*, Volume 20, Issue 3, August 2024. doi:10.46298/lmcs-20(3:14)2024.

2 Luca Aceto, Antonis Achilleos, Duncan Paul Attard, Léo Exibard, Adrian Francalanza, and Anna Ingólfsdóttir. A monitoring tool for linear-time μHML. *Sci. Comput. Program.*, 232:103031, 2024. doi:10.1016/J.SCICO.2023.103031.

3 Luca Aceto, Antonis Achilleos, Adrian Francalanza, Anna Ingólfsdóttir, and Sævar Örn Kjartansson. Determinizing monitors for HML with recursion. *J. Log. Algebraic Methods Program.*, 111:100515, 2020. doi:10.1016/J.JLAMP.2019.100515.

4 Luca Aceto, Antonis Achilleos, Adrian Francalanza, Anna Ingólfsdóttir, and Karoliina Lehtinen. Adventures in monitorability: from branching to linear time and back again. *Proc. ACM Program. Lang.*, 3(POPL):52:1–52:29, 2019. doi:10.1145/3290365.

5 Luca Aceto, Antonis Achilleos, Adrian Francalanza, Anna Ingólfsdóttir, and Karoliina Lehtinen. The best a monitor can do. In Christel Baier and Jean Goubault-Larrecq, editors, *29th EACSL Annual Conference on Computer Science Logic, CSL 2021, January 25-28, 2021, Ljubljana, Slovenia (Virtual Conference)*, volume 183 of *LIPIcs*, pages 7:1–7:23. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.CSL.2021.7.

6 Luca Aceto, Antonis Achilleos, Adrian Francalanza, Anna Ingólfsdóttir, and Karoliina Lehtinen. An operational guide to monitorability with applications to regular properties. *Softw. Syst. Model.*, 20(2):335–361, 2021. doi:10.1007/s10270-020-00860-z.

7 Luca Aceto, Ian Cassar, Adrian Francalanza, and Anna Ingólfsdóttir. On Runtime Enforcement via Suppressions. In *CONCUR*, volume 118 of *LIPIcs*, pages 34:1–34:17, 2018. doi:10.4230/LIPICS.CONCUR.2018.34.

8 Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie J. Hendren, Sascha Kuzins, Ondrej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding trace matching with free variables to AspectJ. In Ralph E. Johnson and Richard P. Gabriel, editors, *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA*, pages 345–364. ACM, 2005. doi:10.1145/1094811.1094839.

9 Bowen Alpern and Fred B. Schneider. Recognizing safety and liveness. *Distributed Comput.*, 2(3):117–126, 1987. doi:10.1007/BF01782772.

10 Krzysztof R. Apt and Gordon D. Plotkin. Countable nondeterminism and random assignment. *J. ACM*, 33(4):724–767, 1986. doi:10.1145/6490.6494.

11 Howard Barringer, Yliès Falcone, Klaus Havelund, Giles Reger, and David E. Rydeheard. Quantified event automata: Towards expressive and efficient runtime monitors. In Dimitra Giannakopoulou and Dominique Méry, editors, *FM 2012: Formal Methods - 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings*, volume 7436 of *Lecture Notes in Computer Science*, pages 68–84. Springer, 2012. doi:10.1007/978-3-642-32759-9_9.

12 Howard Barringer, Allen Goldberg, Klaus Havelund, and Koushik Sen. Rule-based runtime verification. In Bernhard Steffen and Giorgio Levi, editors, *Verification, Model Checking, and Abstract Interpretation, 5th International Conference, VMCAI 2004, Venice, Italy, January 11-13, 2004, Proceedings*, volume 2937 of *Lecture Notes in Computer Science*, pages 44–57. Springer, 2004. doi:10.1007/978-3-540-24622-0_5.

13 Howard Barringer, David E. Rydeheard, and Klaus Havelund. Rule systems for run-time monitoring: From Eagle to RuleR. In Oleg Sokolsky and Serdar Tasiran, editors, *Runtime Verification, 7th International Workshop, RV 2007, Vancouver, Canada, March 13, 2007, Revised Selected Papers*, volume 4839 of *Lecture Notes in Computer Science*, pages 111–125. Springer, 2007. doi:10.1007/978-3-540-77395-5_10.

14 Ezio Bartocci, Yliès Falcone, Adrian Francalanza, and Giles Reger. Introduction to Runtime Verification. In *Lectures on Runtime Verification: Introductory and Advanced Topics*, volume 10457 of *LNCS*, pages 1–33. Springer, 2018. doi:10.1007/978-3-319-75632-5_1.

15 David A. Basin, Felix Klaedtke, and Samuel Müller. Policy monitoring in first-order temporal logic. In Tayssir Touili, Byron Cook, and Paul B. Jackson, editors, *Computer Aided Verification, 22nd International Conference, CAV2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, volume 6174 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2010. doi:10.1007/978-3-642-14295-6_1.

16 Andreas Bauer, Jan-Christoph Küster, and Gil Vegliach. From propositional to first-order monitoring. In Axel Legay and Saddek Bensalem, editors, *Runtime Verification - 4th International Conference, RV 2013, Rennes, France, September 24-27, 2013. Proceedings*, volume 8174 of *Lecture Notes in Computer Science*, pages 59–75. Springer, 2013. doi:10.1007/978-3-642-40787-1_4.

17 Henrik Björklund and Thomas Schwentick. On notions of regularity for data languages. *Theor. Comput. Sci.*, 411(4-5):702–715, 2010. doi:10.1016/j.tcs.2009.10.009.

18 Mikołaj Bojańczyk. Slightly infinite sets, 2019. URL: https://www.mimuw.edu.pl/~bojan/paper/atom-book.

19 Mikolaj Bojanczyk, Claire David, Anca Muscholl, Thomas Schwentick, and Luc Segoufin. Two-variable logic on data words. *ACM Trans. Comput. Log.*, 12(4):27:1–27:26, 2011. doi:10.1145/1970398.1970403.

20 Benedikt Bollig, Aiswarya Cyriac, Paul Gastin, and K. Narayan Kumar. Model checking languages of data words. In Lars Birkedal, editor, *Foundations of Software Science and Computational Structures*, pages 391–405, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. doi:10.1007/978-3-642-28729-9_26.

21 Julian C. Bradfield and Colin Stirling. Modal mu-calculi. In Patrick Blackburn, J. F. A. K. van Benthem, and Frank Wolter, editors, *Handbook of Modal Logic*, volume 3 of *Studies in logic and practical reasoning*, pages 721–756. North-Holland, 2007. doi:10.1016/S1570-2464(07)80015-2.

22 Randal E. Bryant. Binary decision diagrams. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 191–217. Springer, 2018. doi:10.1007/978-3-319-10575-8_7.

23 Feng Chen and Grigore Rosu. Parametric trace slicing and monitoring. In Stefan Kowalewski and Anna Philippou, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, volume 5505 of *Lecture Notes in Computer Science*, pages 246–261. Springer, 2009. doi:10.1007/978-3-642-00768-2_23.

24 Mads Dam. CTL* and ECTL* as fragments of the modal μ-calculus. *Theoretical Computer Science*, 126(1):77–96, 1994. doi:10.1016/0304-3975(94)90269-0.

25 Mads Dam. Temporal logic, automata and classical theories. In *Proceedings of the 6th European Summer School in Logic, Language and Information (ESSLLI'94)*, August 1994.

26 Marcelo d'Amorim and Grigore Rosu. Efficient monitoring of omega-languages. In Kousha Etessami and Sriram K. Rajamani, editors, *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005, Proceedings*, volume 3576 of *Lecture Notes in Computer Science*, pages 364–378. Springer, 2005. doi:10.1007/11513988_36.

27 Loris D'Antoni. In the maze of data languages. *CoRR*, abs/1208.5980, 2012. arXiv:1208.5980.

28 Normann Decker, Martin Leucker, and Daniel Thoma. Monitoring modulo theories. *Int. J. Softw. Tools Technol. Transf.*, 18(2):205–225, 2016. doi:10.1007/S10009-015-0380-3.

29 Stéphane Demri and Ranko Lazic. LTL with the freeze quantifier and register automata. *ACM Trans. Comput. Log.*, 10(3):16:1–16:30, 2009. doi:10.1145/1507244.1507246.

30 Stéphane Demri, Ranko Lazic, and David Nowak. On the freeze quantifier in constraint LTL: decidability and complexity. *Inf. Comput.*, 205(1):2–24, 2007. doi:10.1016/j.ic.2006.08.003.

31 Rüdiger Ehlers, Sanjit A. Seshia, and Hadas Kress-Gazit. Synthesis with identifiers. In Kenneth L. McMillan and Xavier Rival, editors, *Verification, Model Checking, and Abstract*

Interpretation - 15th International Conference, VMCAI 2014, San Diego, CA, USA, January 19-21, 2014, Proceedings, volume 8318 of *Lecture Notes in Computer Science*, pages 415–433. Springer, 2014. doi:10.1007/978-3-642-54013-4_23.

32 E. Allen Emerson. Temporal and modal logic. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, pages 995–1072. Elsevier and MIT Press, 1990. doi:10.1016/b978-0-444-88074-1.50021-4.

33 Léo Exibard, Emmanuel Filiot, and Pierre-Alain Reynier. Synthesis of data word transducers. *Logical Methods in Computer Science*, 17(1), 2021. URL: https://lmcs.episciences.org/7279.

34 Yliès Falcone, Srdan Krstic, Giles Reger, and Dmitriy Traytel. A taxonomy for classifying runtime verification tools. *Int. J. Softw. Tools Technol. Transf.*, 23(2):255–284, 2021. doi:10.1007/S10009-021-00609-Z.

35 Diego Figueira. *Reasoning on words and trees with data. (Raisonnement sur mots et arbres avec données)*. PhD thesis, École normale supérieure de Cachan, France, 2010. URL: https://tel.archives-ouvertes.fr/tel-00718605.

36 Diego Figueira. Alternating register automata on finite words and trees. *Log. Methods Comput. Sci.*, 8(1), 2012. doi:10.2168/LMCS-8(1:22)2012.

37 Diego Figueira, Piotr Hofman, and Slawomir Lasota. Relating timed and register automata. *Math. Struct. Comput. Sci.*, 26(6):993–1021, 2016. doi:10.1017/S0960129514000322.

38 Diego Figueira, Anirban Majumdar, and M. Praveen. Playing with repetitions in data words using energy games. *Log. Methods Comput. Sci.*, 16(3), 2020. URL: https://lmcs.episciences.org/6614.

39 Bernd Finkbeiner, Felix Klein, Ruzica Piskac, and Mark Santolucito. Temporal stream logic: Synthesis beyond the bools. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I*, volume 11561 of *Lecture Notes in Computer Science*, pages 609–629. Springer, 2019. doi:10.1007/978-3-030-25540-4_35.

40 Radu Grigore, Dino Distefano, Rasmus Lerchedahl Petersen, and Nikos Tzevelekos. Runtime verification based on register automata. In Nir Piterman and Scott A. Smolka, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, volume 7795 of *Lecture Notes in Computer Science*, pages 260–276. Springer, 2013. doi:10.1007/978-3-642-36742-7_19.

41 Jan Friso Groote and Radu Mateescu. Verification of temporal properties of processes in a setting with data. In Armando Martin Haeberer, editor, *Algebraic Methodology and Software Technology, 7th International Conference, AMAST '98, Amazonia, Brasil, January 4-8, 1999, Proceedings*, volume 1548 of *Lecture Notes in Computer Science*, pages 74–90. Springer, 1998. doi:10.1007/3-540-49253-4_8.

42 Klaus Havelund and Doron Peled. Efficient runtime verification of first-order temporal properties. In María-del-Mar Gallardo and Pedro Merino, editors, *Model Checking Software - 25th International Symposium, SPIN 2018, Malaga, Spain, June 20-22, 2018, Proceedings*, volume 10869 of *Lecture Notes in Computer Science*, pages 26–47. Springer, 2018. doi:10.1007/978-3-319-94111-0_2.

43 Klaus Havelund, Giles Reger, Daniel Thoma, and Eugen Zalinescu. Monitoring events that carry data. In Ezio Bartocci and Yliès Falcone, editors, *Lectures on Runtime Verification - Introductory and Advanced Topics*, volume 10457 of *Lecture Notes in Computer Science*, pages 61–102. Springer, 2018. doi:10.1007/978-3-319-75632-5_3.

44 Hsi-Ming Ho, Joël Ouaknine, and James Worrell. Online monitoring of metric temporal logic. In Borzoo Bonakdarpour and Scott A. Smolka, editors, *Runtime Verification - 5th International Conference, RV 2014, Toronto, ON, Canada, September 22-25, 2014. Proceedings*,

volume 8734 of *Lecture Notes in Computer Science*, pages 178–192. Springer, 2014. doi:10.1007/978-3-319-11164-3_15.

45 Michael Kaminski and Nissim Francez. Finite-memory automata. *Theor. Comput. Sci.*, 134(2):329–363, 1994. doi:10.1016/0304-3975(94)90242-9.

46 Marcin Jurdzinski and Ranko Lazic. Alternation-free modal mu-calculus for data trees. In *22nd IEEE Symposium on Logic in Computer Science (LICS 2007), 10-12 July 2007, Wroclaw, Poland, Proceedings*, pages 131–140. IEEE Computer Society, 2007. doi:10.1109/LICS.2007.11.

47 Patrick O'Neil Meredith, Dongyun Jin, Dennis Griffith, Feng Chen, and Grigore Rosu. An overview of the MOP runtime verification framework. *Int. J. Softw. Tools Technol. Transf.*, 14(3):249–289, 2012. doi:10.1007/S10009-011-0198-6.

48 Robin Milner. Is computing an experimental science? *J. Inf. Technol.*, 2(2):58–66, 1987. doi:10.1057/JIT.1987.12.

49 Frank Neven, Thomas Schwentick, and Victor Vianu. Finite state machines for strings over infinite alphabets. *ACM Trans. Comput. Log.*, 5(3):403–435, 2004. doi:10.1145/1013560.1013562.

50 Amir Pnueli and Aleksandr Zaks. PSL model checking and run-time verification via testers. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM 2006: Formal Methods, 14th International Symposium on Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 573–586. Springer, 2006. doi:10.1007/11813040_38.

51 Luc Segoufin. Automata and logics for words and trees over an infinite alphabet. In Zoltán Ésik, editor, *Computer Science Logic, 20th International Workshop, CSL 2006, 15th Annual Conference of the EACSL, Szeged, Hungary, September 25-29, 2006, Proceedings*, volume 4207 of *Lecture Notes in Computer Science*, pages 41–57. Springer, 2006. doi:10.1007/11874683_3.

52 Robert S. Streett and E. Allen Emerson. An automata theoretic decision procedure for the propositional mu-calculus. *Information and Computation*, 81(3):249–264, 1989. doi:10.1016/0890-5401(89)90031-X.

53 Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955. doi:10.2140/pjm.1955.5.285.

54 Moshe Y. Vardi. A temporal fixpoint calculus. In Jeanne Ferrante and Peter Mager, editors, *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego, California, USA, January 10-13, 1988*, pages 250–259. ACM Press, 1988. doi:10.1145/73560.73582.