Monitoring Hyperproperties with Circuits

Luca Aceto[1,3], Antonis Achilleos[1], Elli Anastasiadi[1(✉)], and Adrian Francalanza[2]

[1] ICE -TCS, Department of Computer Science, Reykjavik University, Reykjavik, Iceland
{luca,antonios,elli19}@ru.is
[2] Department of Computer Science, University of Malta, Msida, Malta
adrian.francalanza@um.edu.mt
[3] Gran Sasso Science Institute, L'Aquila, Italy
luca.aceto@gssi.it

Abstract. This paper presents an extension of the safety fragment of Hennessy-Milner Logic with recursion over sets of traces, in the spirit of Hyper-LTL. It then introduces a novel monitoring setup that employs circuit-like structures to combine verdicts from regular monitors. The main contribution of this study is the definition of the monitors and their semantics, as well as a monitor-synthesis procedure from formulae in the logic that yields 'circuit-like monitors' that are sound and violation complete over a finite set of infinite traces.

1 Introduction

The field of runtime verification concerns itself with providing methods for checking whether a system satisfies its intended specification at runtime. This runtime analysis is done through a computing device called a *monitor* that observes the current run of a system in the form of a trace [4,12]. Runtime verification has recently been extended to the setting of concurrent systems [1,5,7,16] with several attempts to specify properties over sets of traces, and to introduce novel monitoring setups [2,6,11]. A centerpiece in this line of work has been the specification logic Hyper-LTL [9]. Intuitively Hyper-LTL allows for existential and universal quantification over a set of traces (which describes the set of observed system runs). The properties over one trace are stated in LTL, with free trace variables, and then made dependent on properties of other traces via the quantification that binds the trace variables.

We define the linear-time specification logic Hyper-μHML, as a counterpart to Hyper-LTL, building on previous studies of monitorability and monitor

The authors were supported by the projects 'Open Problems in the Equational Logic of Processes' (OPEL) (grant No 196050–051) and 'Mode(l)s of Verification and Monitorability' (MoVeMent) (grant No 217987) of the Icelandic Research Fund, and 'Runtime and Equational Verification of Concurrent Programs' (ReVoCoP) (grant No 222021), of the Reykjavik University Research Fund. Luca Aceto's work was also partially supported by the Italian MIUR PRIN 2017 project FTXR7S IT MATTERS 'Methods and Tools for Trustworthy Smart Systems'.

© IFIP International Federation for Information Processing 2022
Published by Springer Nature Switzerland AG 2022
M. R. Mousavi and A. Philippou (Eds.): FORTE 2022, LNCS 13273, pp. 1–10, 2022.
https://doi.org/10.1007/978-3-031-08679-3_1

synthesis for μHML [1,13], which are necessary for the kind of correctness and complexity guarantees we aim to achieve in this work. However, just like Hyper-LTL, Hyper-μHML can define dependencies over different traces, which intuitively causes extra delays in the processing of traces as the properties observed on one of them can impact what is expected for another. For example, if a property requires that an event of a trace is compared against an event occurring in all other traces then the processing cost of this event becomes dependent on the number of traces. In this approach, we keep the processing-at-runtime cost (as defined in [17]) minimal by restricting the type of properties verified to a natural fragment of Hyper-μHML, but applying no assumptions on the system under scrutiny. This comes in contrast with the existing research, where the runtime verification of such properties is dealt with via a plethora of modifications and assumptions made over the monitoring setup, such as being able to restart an execution or having access to all executions of a system.

Our monitor setup is engineered for the studied fragment of the specification language, by utilizing circuit-like structures to combine verdicts over different traces. The fragment of the logic restricts the amount of quantification that can be applied to the properties of individual traces and thus limits the dependencies between them. This naturally induces circuits with monitors from [1] as input nodes and simple kinds of gates at the higher levels, with the resulting structure having constant depth with respect to the corresponding formula, which is considered efficient in the field of parallel computation [14]. Thus, each step taken by such a monitor in response to an event of the system under scrutiny takes constant time, which makes the monitors 'real time' in the sense of [17].

2 The Logic

Our logic is defined in the style of Hyper-LTL as presented in [9]. The quantification among traces remains the same, but the language in which local trace properties are stated is μHML. We consider the following restriction to a multi-trace sHML logic (the *safety* fragment of μHML [1]), with no alternating quantifiers, called HYPER^1-sHML. We can similarly define the cHML (co-safety) fragment, and the HML fragment.

Definition 1. *Formulae in* HYPER^1-sHML *are constructed by the following grammar:*

$$\varphi \in \text{HYPER}^1\text{-sHML} ::= \exists_\pi \psi \quad | \ \forall_\pi \psi \quad | \ \varphi \sqcup \varphi \quad | \ \varphi \sqcap \varphi$$

where ψ stands for a formula in sHML *and π is a trace variable from an infinite suppy of trace variables $\mathcal{V}$. $\sqcup$ and $\sqcap$ stand for the regular $\vee$ and $\wedge$ boolean connectives, only usable at the top syntax level. Although the syntactic distinction is cosmetic, it allows us to keep the synthesis function in Definition 4 clearer.*

Semantics. The semantics of Hyper-μHML is given over a finite set of infinite traces T over ACT and it is a natural extension of the linear-time semantics

of μHML. The existential and universal quantification happens via the trace variable π which ranges over the traces in T. The extension of the μHML linear-time semantics from [1] to the Hyper-μHML semantics is done in the style of Hyper-LTL. This semantics applies to HYPER^1-sHML, which is a fragment of Hyper-μHML. We only consider *closed* formulae in HYPER^1-sHML and for these we use the standard notation $T \models \varphi$ to mean that a set of traces T satisfies φ (and similarly for $T \not\models \varphi$).

Example 1. The HYPER^1-sHML formula $\forall_\pi[a]\mathtt{ff} \sqcap \exists_\pi[b](max\ x.([a]\mathtt{ff} \wedge [b]x))$, over the set of actions $\{a, b\}$, states that for any set of traces T, none of the traces in T start with a, and $b^\omega \in T$.

3 The Monitors

The intuition behind our monitor design is the following (we recommend following this intuition along with the example given in Fig. 1). Over a finite set of traces T we instrument a circuit-like structure. Each trace $t \in T$ is assigned a fixed set of regular monitors that correspond to the properties in sHML to be verified. These regular monitors are connected with simple gates which evaluate to *yes*, *no* or *end* based on the verdicts produced by their associated regular monitors. Once some of these gates start evaluating to verdicts, they communicate with more complex gates, connected in a circuit-like graph, which propagate input verdicts though logic operations until the root node of the circuit reaches a verdict as well. The formal definition of a circuit monitor is given in the style of computational complexity circuits [18, Definition 1.10].

Definition 2. *The language* CMON$_k$ *of k-ary monitors, for* $k > 0$ *is given through the following grammar:*

$$M \in \text{CMON}_k ::= \bigvee[m]_k \mid \bigwedge[m]_k \mid M \vee M \mid M \wedge M$$
$$m ::= \quad yes \mid no \mid end \mid a.m,\ a \in \text{ACT} \mid m + n \mid rec\ x.m \mid x$$

CMON *is the collection of infinite sequences* $(M_i)_{i\in\mathbb{N}}$ *of terms that are generated by substituting* $k = i, \forall i \in \mathbb{N}$*, in a term* M *in* CMON$_k$.

We use $M, M' \ldots$ to denote the monitors (infinite sequences of terms generated by the first line of this grammar), and refer to them as circuit monitors, and $m_1, m_2 \ldots$ to denote the regular monitors described by the second line. The notation $[m]_k$ corresponds to the parallel dispatch of k identical regular monitors m, where $k = |T|$, with $T = \{t_1, \ldots, t_k\}$.

Given a monitor $M \in$ CMON, we will call each syntactic sub-monitor of M a gate. For example, we have inductively that over the monitor $M' \vee M''$ we have the gates $M' \vee M''$ and all gates contained in monitors M', and M'', while for the monitor $\bigvee[m]_k$ we have the gates $\bigvee[m]_k$ and gates $m_{[i]}$ for $i \in \{1, \ldots, k\}$. For $M \in$ CMON we define a set of *program variables* G_M, where one variable $g_{M'}$ is assigned to each gate M' of M.

For readability purposes we will be omitting the naming g of the program variables and call them by the name of the gate they represent. We use $m_{[i]}$ to mean the regular monitor m instrumented over the trace t_i. It is important here to see that $g_{m_{[i]}}$ will be the *name* of the gate assigned to one such monitor and stays unchanged while the actual monitor advances its computation as trace events are read. This will be clarified later, through the instrumentation rules.

A program variable related to gate M, can be assigned the following values: *yes*, *no*, *end*, and j, with $j \in \{0, \ldots, 2^{(\ell+1)}-1\}$, ℓ being the number of immediate syntactical sub-monitors of gate M. Number j is encoded in binary, and is used to carry the information of which sub-gates have given some verdict (this means that the encoding of j has $\ell+1$ bits). The value of the $\ell+1$-th bit of j is reserved to encode that one of the sub gates has outputted an *end*. The information that j carries is very important for the evaluation of a gate, as often this evaluation depends on the verdicts of more that one sub-gate, as well as what these verdicts are (see Fig. 1). A variable g_m can only take the values *yes*, *no* and *end*, produced by the relevant monitor instrumented over a trace.

A ***configuration*** of monitor M is an array s_M containing a value for all program variables g of M. We denote the set of all configurations for a monitor M as $\mathcal{S}_M$. We use the notation $s[M\backslash i]$ to denote the update of a configuration s where gate M stores some value j to one where the i-th coordinate of j is 0, while all other variables have the value they had in configuration s. Similarly, we use the notation $s[M\backslash end_i]$ to refer to a configuration where the update $s[M\backslash i]$ has taken place *and* the value of the $\ell+1$-th bit of j is set to 1, and we also use the notation $s[^{v}/_{M}]$ with $v \in \{yes, no, end\}$, to mean a configuration where the value of the variable for gate M is updated to v,

All gate variables in a circuit monitor are initialized to $2^{\ell}-1$ (a sequence of ℓ-many zeros), to represent that all sub-gates are waiting to give some output and $s_{M_{init}}$ stands for the initial configuration of M. Since M is a family of circuits, we have that the initial configuration of each monitor M_i in the family corresponds to a different initial configuration $s_{M_{i-init}}$.

Example 2. In Fig. 1, we give an example of a circuit monitor and its evaluation.

Semantics. The semantics of a regular monitors is as presented in [1]. Each regular monitor corresponds to an LTS, and a transition labeled with $a \in \text{Act}$ corresponds to a regular monitor observing the event a when instrumented with a system p that produces it. The semantics of a circuit monitor is given as a transition relation $\rightarrow \subseteq \mathcal{S}_M \times \mathcal{S}_M$ and the instrumentation $\triangleleft$ takes place over a set of regular monitors $\overrightarrow{m}$ instrumented over a set of traces T, denoted $M(T)$.

We define $M(T) := s_{M_{|T|-init}} \triangleleft \overrightarrow{m}_{[i]} \triangleleft T$, where $\overrightarrow{m}$ is the set of regular monitors that occur in M, and $\overrightarrow{m}_{[i]}$ is $\overrightarrow{m}$, instrumented over the trace $t_i \in T$. When m is a regular monitor then $\triangleleft$ stands for the existing instrumentation relation from [1]. The transition and instrumentation relations are defined as the least ones that satisfy the axioms and rules in Fig. 2. Due to lack of space, we only include the rules giving the semantics of the $\bigvee[m]_k$ monitor. Those for the other operators follow the same structure. The proof in Appendix A could help with the understanding of the more intricate instrumentation rules.

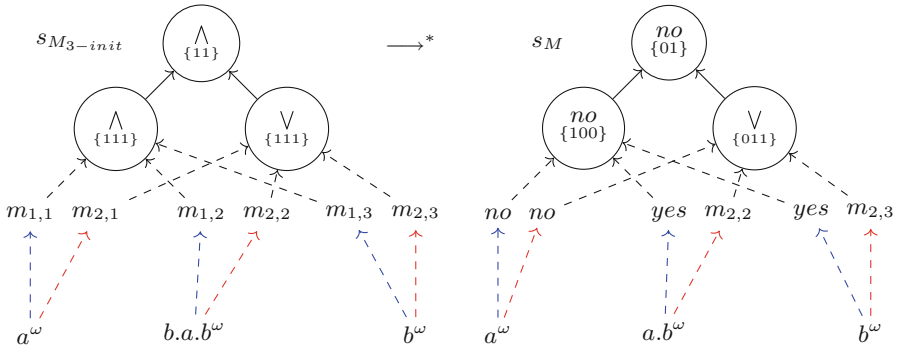

Fig. 1. The circuit monitor for the formula from Example 1, over $T = \{a^\omega, b.a.b^\omega, b^\omega\}$.

Monitor semantics:

$$\frac{s[m_{[i]}] = yes}{s \to s[^{yes}/_{\bigvee[m]_k}]} \qquad \frac{s[m_{[i]}] = no}{s \to s[\bigvee[m]_k \backslash i]} \qquad \frac{s[m_{[i]}] = end}{s \to s[\bigvee[m]_k \backslash end_i]}$$

$$\frac{s[\bigvee[m]_k] = 0}{s \to s[^{no}/_{\bigvee[m]_k}]} \qquad \frac{s[\bigvee[m]_k] = 2^k}{s \to s[^{end}/_{\bigvee[m]_k}]}$$

Instrumentation:

$$\frac{m \xrightarrow{\tau} m'}{m \triangleleft t \xrightarrow{\tau} m' \triangleleft t} \qquad \frac{m \xrightarrow{a} m'}{m \triangleleft a.t \xrightarrow{a} m' \triangleleft t} \qquad \frac{\forall j \in \{1, \dots r\},\ m_{j[i]} \triangleleft t \xrightarrow{a} m'_{j[i]} \triangleleft t'}{s \triangleleft (\overrightarrow{m} \triangleleft T) \to s \triangleleft (\overrightarrow{m}[^{m'_{j[i]}}/_{m_{j[i]},\ \forall j}] \triangleleft T[t'/t])}$$

$$\frac{s \to s'}{s \triangleleft (\overrightarrow{m} \triangleleft T) \to s' \triangleleft (\overrightarrow{m} \triangleleft T)} \qquad \frac{s \triangleleft (\overrightarrow{m} \triangleleft T) \to s \triangleleft (\overrightarrow{m}[^{v}/_{n_{j[i]}}] \triangleleft T[t'/t])}{s \triangleleft (\overrightarrow{m} \triangleleft T) \to s[^{v}/_{g_{m_{j[i]}}}] \triangleleft (\overrightarrow{m}[^{v}/_{n_{j[i]}}] \triangleleft T[t'/t])}$$

Fig. 2. Operational semantics of processes in CMON.

A monitor is required to be *correct* with respect to some specification formula φ. The notions of correctness we use in this work are defined below.

Definition 3. *Given a monitor $M \in$ CMON, and a set of traces T.*

- *M **rejects** T (resp. **accepts** T) denoted $rej(M,T)$ (resp. $acc(M,T)$) iff $M(T) \to^* s \triangleleft \overrightarrow{n} \triangleleft T'$ for some s, $\overrightarrow{n}$, T', where $s[M] = no$ (resp. $s[M] = yes$).*
- *Given a formula $\varphi \in$ Hyper-μHML, M is **sound** for φ if $\forall T$, $acc(M,T) \implies T \models \varphi$, and $rej(M,T) \implies T \not\models \varphi$.*
- *M is **violation complete** for φ if $\forall T$, $T \not\models \varphi \implies rej(M,T)$.*

Synthesis: Given a formula φ in HYPER^1-SHML, We synthesize a circuit monitor M through the following recursive function $Syn(-)$: HYPER^1-SHML $\to$ CMON.

Definition 4 (Circuit Monitor Synthesis).

$$Syn(\exists_\pi \varphi) = \bigvee [m(\varphi)]_k \qquad Syn(\forall_\pi \varphi) = \bigwedge [m(\varphi)]_k$$
$$Syn(\varphi_1 \sqcup \varphi_2) = Syn(\varphi_1) \vee Syn(\varphi_2) \qquad Syn(\varphi_1 \sqcap \varphi_2) = Syn(\varphi_1) \wedge Syn(\varphi_2)$$

Where $m(-)$ is the monitor synthesis function for sHML *defined in [1].*

Proposition 1. *Given a formula φ in* Hyper^1-sHML*, we have $Syn(\varphi)$ is a sound and violation-complete monitor for φ.*

Proof. The proof is by induction on the structure of φ. We present here a characteristic case and give more details for some of them in the Appendix A. Assume that $\varphi = \exists_\pi \psi$, with $\psi \in$ sHML and that we have a set of traces T s.t. $T \not\models \varphi$. From the semantics of Hyper^1-sHML, we have that $t_i \not\models \psi$, for all traces t_i in T. However $\psi \in$ sHML and thus from [1] we get that m_ψ is a violation complete monitor for ψ. This means that for all $t_i \in T$, there exist $t_i' \in$ Act* and $t_i'' \in$ Act$^\omega$, such that $t_i = t_i'.t_i''$, such that the monitor m_ψ rejects t_i'.

From the rules in Fig. 2 we see that each gate $g_{m_{\psi[i]}}$ will reach the value *no* as enough events over the trace t_i' will occur. I.e. $s_M \triangleleft \overrightarrow{m_{\psi_{[i]}}} \triangleleft T \rightarrow^* s_M \triangleleft \overrightarrow{m}_{[i]}[^{no}/_{m_{[i]}}] \triangleleft T[^{t_i''}/_{t_i}]$, witch propagates to the evaluation of $g_{m_{[i]}}$ to *no*, for all i. We now study the transitions $s_M[^{no}/_{g_{m_\psi[i]}}]$ since those can be then composed with this instrumentation via the fourth instrumentation rule. Applying the SOS rules yields that the update $\backslash i$ takes place for all i at the gate $\bigvee[m]_k$ which means that the value of j stored in it becomes 0. This finally yields that the value of the final gate $\bigvee[m]_k$ becomes 0, i.e. $s_M[^{no}/_{g_{m_{[i]}}}\ \forall i] \rightarrow s_M[^{no}/_{\bigvee[m]_k}]$. Since this transition can be composed with the discussed instrumentation we have that $s_M \triangleleft \overrightarrow{m_{\psi_{[i]}}} \triangleleft T \rightarrow s_M[^{no}/_{g_{\bigvee[m_\psi]_{[i]}}}] \triangleleft \overrightarrow{n} \triangleleft T'$ for some $\overrightarrow{n}$ and T and we are done. □

3.1 Runtime Costs

The monitor synthesis in Definition 4 provides a family of circuits that can be instrumented appropriately on an arbitrary set of traces to analyze the events occurring in them. Ideally, the runtime cost of monitoring resulting from our constructions should be bounded by a constant that does not depend on the parameters of the system (such as the number of available traces, or of the events observed so far) [17]. In this way, if a monitor is launched along with the system components, it will only induce a feasible computational overhead.

We already know that the regular monitors instrumented with individual traces analyze the system events they observe with a constant overhead [13]. Regarding the computational cost of the circuit part, since we are given k many traces, it must be that the necessary computation performed from a circuit monitor can be performed in parallel, distributed over the components that produced the traces in the first place. This means that we can only concern ourselves with the *circuit complexity* [18] of a given monitor, which encapsulates the parallel processing power necessary for its evaluation.

We now observe the synthesis function. There, a formula φ in HYPER^1-SHML will be turned into a family of circuit monitors where, for each connective of the original formula φ, the output monitor increases in size based on the size for the monitors of the sub-formulae of φ. However, for each connective of the formula, the *depth* of the circuit is only increased by 1 which means that the output circuit monitor has a depth bounded by the size of the formula φ. Since the gates of the output monitor can have either a fixed amount of sub-gates $(\vee, \wedge)$, or k many $(\bigvee, \bigwedge)$, we have that the output circuit is in the complexity class AC^0 [18]. Thus, the monitor only adds a constant computational overhead when executed over the computational resources of the distributed components of the system.

4 Conclusion and Future Work

We expect that the fragment HYPER^1-SHML is maximal with respect to violation completeness, which means that any monitor in CMON is monitoring for a formula in HYPER^1-SHML. However, the ultimate goal of this work is to extend the collection of monitorable properties by allowing alternating quantifiers in the syntax. This is a very important aspect of any work in this field, as the more interesting hyperproperties, such as the property "at all times, if one trace encounters the event p then all traces do so as well" which is a necessary component for the expression of properties such as noninference [8,15], require alternation of quantifiers.

A way to tackle this would be to project such properties into the HYPER^1-SHML fragment. However this procedure is not formally yet defined, or trivial and one could argue that since every hyperproperty has been shown ([9]) to be the intersection of a liveness and a safety hyperproperty, (and since liveness and safety properties are widely accepted as independent [3]), an elimination of alternating quantifiers can only take place in very few cases. Thus, our main purpose is to extend the logic and the consequent monitors in order to express and monitor for the most general class of such properties. The main objective of the logical fragment we give here is to establish a formal baseline which we will attempt to extend in future work.

Our approach to an extension would be to allow a notion of synchronization rounds among the regular monitors (or equivalently a round of communication). This would enable more complex dependencies between traces, as now the properties required of a given trace can be impacted by the state of the ones monitored for on a different one. However, the analysis of communications among the monitors is a complicated extension, as their exact content plays a significant role to our insight over the system, as well as the processing at runtime cost. We plan to implement this therefore by utilizing dynamic epistemic logic [10] in order to perform this extension formally and soundly.

A Appendix: Cases for the Proof of Violation Completeness

Here we give some more insight on the remaining cases of the violation completeness proof. First we highlight that the second base case of our proof, for formulae of the form $\forall_\pi \psi$ is completely analogous to the one we give and thus omitted.

We will here give an important lemma necessary for analyzing both remaining cases, and then present the high level details for the case of $\sqcap$. The intuition of the importance of the lemma is that the monitors $Syn(\varphi_1)$ and $Syn(\varphi_2)$ should not have their computation affected from the fact that they are run in parallel over a set of traces T.

Lemma 1. *If*

- $s_{M_1} \lhd \overrightarrow{m_1}[i] \lhd T \rightarrow s'_{M_1} \lhd \overrightarrow{m_1}[i]' \lhd T'$, *and*
- $s_{M_2} \lhd \overrightarrow{m_2}[i] \lhd T \rightarrow s'_{M_2} \lhd \overrightarrow{m_2}[i]' \lhd T'$

then

- $s_{M_1 \vee M_2} \lhd \overrightarrow{m_{12}}[i] \lhd T \rightarrow s'_{M_1 \wedge M_2} \lhd \overrightarrow{m_{12}}[i]' \lhd T'$, *and*
- $s_{M_1 \wedge M_2} \lhd \overrightarrow{m_{12}}[i] \lhd T \rightarrow s'_{M_1 \wedge M_2} \lhd \overrightarrow{m_{12}}[i]' \lhd T'$,

where $\overrightarrow{m_{12}} = \overrightarrow{m_2} \cup \overrightarrow{m_2}$ *and* $\overrightarrow{m_{12}}' = \overrightarrow{m_2}' \cup \overrightarrow{m_2}'$ *respectively.*

Proof. We note here that a configuration for $s_{M_1 \vee M_2}$ is identical to one for $s_{M_1 \wedge M_2}$ except the root variable, as all other variables they both contain are $s'_{M_1} \cup s'_{M_2}$.

The key aspect of this proof is the third rule of the instrumentation relation. There we can see that in order for a configuration instrumented over a set of regular monitors, instrumented over a set of traces, can only advance its computation, if all monitors instrumented over the same trace progress with their computation synchronously by reading the next trace event.

Thus, form the assumptions of this lemma we get that for all $j = \{1, \dots r\}$, where r is the total amount of different regular monitors occurring in M_1 and M_2 the premise of our rule is satisfied and thus the cumulative configuration of variables amounting for the union of variables of the two circuit monitors M_1 and M_2 (including the root variable), can perform the necessary transition to the new state, where all regular monitors (those both from M_1 and M_2) assigned to trace t_i have processed the event a, and we are done. □

Having the above lemma streamlines our inductive step for the rest of the cases. Assuming a non-base-case formula in HYPER^1-sHML we can clearly see that it must be of the form $\varphi = \varphi_1 \sqcap \varphi_2$ or $\varphi = \varphi_1 \sqcap \varphi_2$. We only analyze one of the two cases as they are symmetrical. For any set of traces T, such that $T \not\models \varphi$, from the semantics of HYPER^1-sHML, we have that $T \not\models \varphi_1$ and $T \not\models \varphi_2$. Since the synthesized monitor for $\varphi_1 \sqcap \varphi_2$ can reach a configuration where the values of the gates for $Syn(\varphi_1)$ and $Syn(\varphi_2)$ are the same as they would be

for the individual monitors instrumented over T, and by inductive hypothesis (which guarantees that $Syn(\varphi_1)$ and $Syn(\varphi_2)$ are violation-complete) we have necessary conclusion by combining the two negative verdicts of the individual monitors via the semantics. □

References

1. Aceto, L., Achilleos, A., Francalanza, A., Ingólfsdóttir, A., Lehtinen, K.: Adventures in monitorability: from branching to linear time and back again. Proc. ACM Program. Lang. POPL **3**(52), 1–29 (2019)
2. Agrawal, S., Bonakdarpour, B.: Runtime verification of k-safety hyperproperties in HyperLTL. In: IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27–July 1, 2016, pp. 239–252. IEEE Computer Society (2016)
3. Alpern, B., Schneider, F.B.: Recognizing safety and liveness. Distrib. Comput. **2**(3), 117–126 (1987)
4. Bartocci, E., Falcone, Y., Francalanza, A., Reger, G.: Introduction to runtime verification. In: Bartocci, E., Falcone, Y. (eds.) Lectures on Runtime Verification. LNCS, vol. 10457, pp. 1–33. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-75632-5_1
5. Bocchi, L., Honda, K., Tuosto, E., Yoshida, N.: A theory of design-by-contract for distributed multiparty interactions. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010. LNCS, vol. 6269, pp. 162–176. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15375-4_12
6. Bonakdarpour, B., Finkbeiner, B.: The complexity of monitoring hyperproperties. In: 31st IEEE Computer Security Foundations Symposium, CSF 2018, Oxford, United Kingdom, July 9–12, 2018, pp. 162–174. IEEE Computer Society (2018)
7. Cassar, I., Francalanza, A., Mezzina, C.A., Tuosto, E.: Reliability and fault-tolerance by choreographic design. In: Francalanza, A., Pace, G.J. (eds.), Proceedings Second International Workshop on Pre- and Post-Deployment Verification Techniques, PrePost@iFM 2017, Torino, Italy, 19 September 2017, vol. 254 of EPTCS, pp. 69–80 (2017)
8. Clarkson, M.R., Finkbeiner, B., Koleini, M., Micinski, K.K., Rabe, M.N., Sánchez, C.: Temporal logics for hyperproperties. In: Abadi, M., Kremer, S. (eds.) POST 2014. LNCS, vol. 8414, pp. 265–284. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-54792-8_15
9. Clarkson, M.R., Schneider, F.B.: Hyperproperties. J. Comput. Secur. **18**(6), 1157–1210 (2010)
10. van Ditmarsch, H., van der Hoek, W., Kooi, B.: Dynamic Epistemic Logic, 1st edn. Springer, Dordrecht (2007). https://doi.org/10.1007/978-1-4020-5839-4
11. Finkbeiner, B., Hahn, C., Stenger, M., Tentrup, L.: Monitoring hyperproperties. Formal Methods Syst. Des. **54**(3), 336–363 (2019)
12. Francalanza, A., et al.: A foundation for runtime monitoring. In: Lahiri, S., Reger, G. (eds.) RV 2017. LNCS, vol. 10548, pp. 8–29. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67531-2_2
13. Francalanza, A., Aceto, L., Ingólfsdóttir, A.: Monitorability for the hennessy-milner logic with recursion. Formal Methods Syst. Des. **51**(1), 87–116 (2017)
14. Håstad, J.: Computational Limitations of Small-Depth Circuits, vol. 53. MIT Press, Cambridge (1987)

15. McLean, J.: A general theory of composition for a class of "possibilistic" properties. IEEE Trans. Softw. Eng. **22**(1), 53–67 (1996)
16. Mezzina, C.A., Pérez, J.A.: Causally consistent reversible choreographies: a monitors-as-memories approach. In: Proceedings of the 19th International Symposium on Principles and Practice of Declarative Programming, PPDP 2017, pp. 127–138, New York, Association for Computing Machinery (2017)
17. Rabin, M.O.: Real time computation. Israel J. Math. **1**(4), 203–211 (1963)
18. Vollmer, H.: Introduction to Circuit Complexity - A Uniform Approach. Texts in Theoretical Computer Science. An EATCS Series. Springer, New York (1999)