Decentralized LTL Enforcement

Florian Gallay Yliès Falcone
Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, Laboratoire d'Informatique de Grenoble, 38000 Grenoble, France
florian.gallay1@etu.univ-grenoble-alpes.fr, ylies.falcone@univ-grenoble-alpes.fr

We consider the runtime enforcement of Linear-time Temporal Logic formulas on decentralized systems with no central observation point nor authority. A so-called enforcer is attached to each system component and observes its local trace. Should the global trace violate the specification, the enforcers coordinate to correct their local traces. We formalize the decentralized runtime enforcement problem and define the expected properties of enforcers, namely soundness, transparency and optimality. We present two enforcement algorithms. In the first one, the enforcers explore all possible local modifications to find the best global correction. Although this guarantees an optimal correction, it forces the system to synchronize and is more costly, computation and communication wise. In the second one, each enforcer makes a local correction before communicating. The reduced cost of this version comes at the price of the optimality of the enforcer corrections.

1 Introduction

Runtime verification [1, 14] is the collection of theories, techniques, and tools dedicated to the verification of system executions against a formal specification. *Runtime enforcement* (cf. [15, 17]) extends runtime verification and consists in using runtime *enforcers* to *ensure* the absence of violation to the specification. The specification is formalized for instance as a Linear-time Temporal Logic (LTL) formula [27]. Usually, the system is seen as a black box; only its execution is observable (not its implementation). The execution is abstracted as a sequence of events where each event contains the set of relevant atomic propositions that hold on the system state. In (centralized) enforcement, the sequence of events, called trace, is fed to one (central) enforcer which transforms it and outputs a sequence that does not violate the property. Usually, enforcers must be *sound*, *transparent* and *optimal*, that is, their output trace should not violate the property, they should only alter the execution if needed (i.e. to prevent property violations), and the alteration should be minimal, respectively.

Motivation and challenges. We consider decentralized systems, that is systems with no central observation point nor authority but which are instead composed of several components, each producing a local trace. Decentralized systems abound (e.g., multithreaded processors, drone swarms, decentralized finance), some of which can have safety critical properties to be ensured. It is desirable to define enforcement techniques for decentralized systems so as to ensure their desired properties. In the decentralized setting, enforcers should coordinate and modify their local traces in such a way that the global trace respects the enforced property.

Approach overview. We address the problem of enforcing LTL formulas on decentralized systems. We start by defining the runtime enforcement problem in the decentralized setting. Then, we define our enforcement algorithms, which intuitively proceed as follows. Upon each new event σ emitted by the system, using LTL expansion laws [27], we transform the formula to be enforced at the current timestamp into what we refer to as a *temporal disjunctive normal form*, where each disjunct is composed of a present and future obligation formula separated. The enforcers can then evaluate the present obligations and alter their local observation if needed, i.e. when outputting σ would violate it. We note that our enforcers only

To appear in EPTCS.

evaluate the present obligations in the disjuncts. After their evaluations with the current event, a subset of the disjuncts will have their present obligations different from $\bot$. Only the future obligations of these disjuncts are kept for the next timestamp. This strategy spares the rewriting of the future obligations of the discarded disjuncts. When σ needs to be corrected, the enforcers keep track of possible corrections of σ and of the associated formulas. As the system is decentralized, each enforcer can only observe some atomic propositions of the system. Therefore, the enforcers update the associated formulas with their local observations. Then, they send it to another enforcer that will, in turn, do the same until the present obligations are entirely evaluated. In our first algorithm, the formula is evaluated with every possible event over the set of atomic propositions of the system. This allows the enforcers to find the best possible correction if needed. Through communication, they will naturally build the entire set of possible events (the update can be seen as the exploration of a tree whose leaves represent every possible assignment of the atomic propositions). After each local update, each enforcer garbage collects the events that cannot be extended to a viable correction of σ. Then, once the formula has been entirely evaluated, the output event is so that soundness, transparency, and optimality are preserved. Our second algorithm proceeds similarly: the only difference is that, instead of exploring the entire set of possible events, each enforcer makes a local decision before sending the function to another enforcer. This decision consists in choosing one event from the domain (w.r.t. soundness and transparency as well) and only sending this one in order to prevent the exponential growth of the domain. In both cases, the formula to be enforced in the next timestamp is built after choosing the output event. Since enforcers make (optimal) local decisions, optimality of the global event is not guaranteed by the second algorithm using the future obligations associated with the emitted event.

Related work. This paper is at the intersection of two topics, namely *decentralized monitoring* and *runtime enforcement*. In decentralized monitoring (cf. [18] for an overview), a lot of work has been done on the verification of decentralized systems for several specifications languages such as LTL and finite-state automata. These research efforts differ mainly in the assumptions they make on the underlying system. Similarly to this paper, some existing approaches to decentralized monitoring (e.g., [3, 6]) are based on formula rewriting [29]; the specification is usually represented as an LTL formula or in an extension of LTL like MTL [30] and then rewritten and simplified until a verdict can be emitted. Other approaches focus on monitoring distributed systems and tackle the problem of global predicate detection [24, 25] or of fault tolerance for monitors [5], that is, reaching consensus with monitors that are subject to faults. The aforementioned work performs decentralized monitoring on *centralized* specifications. In [9, 11], the focus is on monitoring decentralized specifications that is, multiple interdependent specifications that apply to separate parts of the system.

The above approaches are dedicated to verification in that they focus on determining a verdict but do not consider at all 1) what should be done when the property is violated and 2) what can be done to prevent violations. Runtime enforcement (cf. [15]) approaches try to prevent violations during the execution of the system. This topic has also seen a lot of research efforts on modeling and synthesizing enforcement monitors from several specifications formalisms for discrete-time [16, 12, 7] and timed properties [26, 13, 17] and even stochastic systems [23]. To the best of our knowledge, the only enforcement approaches for decentralized systems are [20, 21], respectively tailored to artifact documents and robotic swarms. However, in this paper, we formally define the decentralized runtime enforcement problem in a generic manner and provide two generic algorithms. Finally, we note that the setting of our approach also differs from the one in runtime enforcement techniques with (uncontrollable) actions/events [2, 28, 22] where system action/events are blocked/buffered in that our monitors instead directly modify the truth value of some atomic propositions of interest.

Outline. Sec. 2 defines preliminary notions. Sec. 3 introduces the *decentralized runtime enforcement* problem. Sec. 4, defines the transformation of the formula allowing the separation between the *present* and *future*. Sec. 5 defines the data structure used to encode the enforcer state. In Sec. 6, we define how the enforcers evaluate the formula. Sec. 7 and 8 present and compare the algorithms based on global and local exploration, respectively. Sec. 9 concludes and outlines some research avenues.

The extended version [19] of this paper contains details about some formula transformations, a complete example of an execution of both algorithms, and proposition proofs.

2 Preliminary Notions

This section introduces some preliminary notions and states the assumptions of our approach.

Decentralized systems. A decentralized system consists of n components $C_1,\dots,C_n$. On each component C_i, $i \in [1\dots n]$, we assume a local set of atomic propositions of interest AP_i. We also assume that $\{AP_1,\dots,AP_n\}$ forms a partition of AP.

Events and traces. An *event* is a set of atomic propositions describing the system state. For an event $\sigma \in 2^{AP}$, when $p \in \sigma$, it means that atomic proposition p holds on the system. We denote the set of all events 2^{AP} as Σ and we call this set the *alphabet*. Similarly, $\Sigma_i = 2^{AP_i}$ denotes the set of local events to component C_i. Note that $\Sigma \neq \bigcup_{i\in[1,n]} \Sigma_i$.

At runtime, each component C_i emits a *local trace* u_i of events from its local set of atomic propositions AP_i. At any timestamp t, the local trace is of the form $u_i(1)\cdot u_i(2)\cdots u_i(t)$ with $\forall t' < t, u_i(t') \in \Sigma_i$ and where $u_i(j)$ represents the j-th local event of C_i. The *global trace* represents the sequence of events emitted by the system as a whole. At any timestamp t, the global trace is of the form: $u = u(1)\cdot u(2)\cdots u(t)$ with $\forall t' < t, u(t') \in \Sigma$ and where $u(j)$ represents the j-th event emitted by the system. It is possible to build the global trace from the local traces: $u = u_1(1)\cup\cdots\cup u_n(1)\cdot u_1(2)\cup\cdots\cup u_n(2)\cdots u_1(t)\cup\cdots\cup u_n(t)$ as well as the local traces from the global trace: $u_i = u(1)\cap AP_i\cdots u(t)\cap AP_i$. The set of all finite traces over an alphabet Σ is denoted as Σ^* whereas the set of all infinite traces over Σ is denoted as Σ^ω. The suffix of a (finite or infinite) trace starting at time t is $w^t = w(t)\cdot w(t+1)\cdots$. The set of all traces is denoted as $\Sigma^\infty = \Sigma^* \cup \Sigma^\omega$.

To measure the differences between two events, we use a distance function that returns the number of atomic propositions with different value between them.

Definition 1 (Distance between events) *Let $\sigma, \sigma'' \in \Sigma$. Function* `distance`: $\Sigma \times \Sigma \times 2^{AP} \to \mathbb{N}$ *is defined as follows:* $\texttt{distance}(\sigma,\sigma'',AP) = \#\{AP \cap (\sigma \cap \overline{\sigma''} \cup \overline{\sigma} \cap \sigma'')\}$*, where the complementary events are taken w.r.t. AP.*

Note that we cannot directly use the Hamming Distance because the events are not represented as strings composed of the atomic propositions or their negations. Instead, an atomic proposition with value $\bot$ is not included in the set.

Linear-time temporal logic on finite traces. The specification of the expected system behavior is formalized using Linear-time Temporal Logic (*LTL*) [27] over the (global) set of atomic propositions AP. We refer to the set of syntactically correct LTL formulas over AP as *LTL*. We assume the reader is familiar with LTL and its operators (Globally (**G**), Eventually (**F**), strong Until (**U**), ...). We denote by $\vDash$ the usual semantic relation between traces and formulas. We say that two formulas φ_1 and φ_2 are

semantically equivalent if for any $w \in \Sigma^\infty$, $w \vDash \varphi_1$ iff $w \vDash \varphi_2$ and we denote this by $\varphi_1 \equiv \varphi_2$. In this paper, we use a finite-trace semantics (from [4]). A finite trace u evaluates to $\top$ (resp. $\bot$) for φ if all its infinite extensions satisfy (resp. do not satisfy) φ. We denote this by $u \in good(\varphi)$ (resp. $u \in bad(\varphi)$). In monitoring and evaluating formulas, we will need to refer to the atomic propositions that have not been evaluated yet in a formula: $\texttt{apFormula}(\varphi) \subseteq AP$ is the set of free atomic propositions occurring on φ.

Normal forms. To use the definition of literals, monomials and normal form with Linear-time temporal logic, we extend them to cover temporal operators: A *literal* is an atomic proposition or the negation of an atomic proposition. A *monomial* is a conjunction of literals and/or of temporal operators applied to any formulas. A formula is in *normal form* if it only contains the operators $\vee, \wedge$ and $\neg$ and/or temporal operators (**X**, **G**, **F**, **U**, **R**) and if negations that are not below a temporal operators are only applied to atomic propositions. Finally, a formula is in *disjunctive normal form (DNF)* if it is a disjunction of monomials. We say that a formula is in *temporal disjunctive normal form (TDNF)* if it is in DNF and each monomial is of the form $\varphi_1 \wedge \mathbf{X}\varphi_2$ where φ_1 only contains propositional logic operators (i.e. it represents a condition on the present).

`map`, `fold`, and `filter`. We shall make use of functions `map`, `fold` and `filter`. Consider two arbitrary types/sets A and B. Function `map` takes as argument a function $f : A \to B$ and a set S containing elements of some type A. It then returns the set $S' = \{f(s) \mid s \in S\}$. Function `filter` takes as argument a predicate p over elements of A and a set S of elements from A and returns the subset of S with elements that satisfy p. Function `fold` takes as arguments a commutative function $f : A \times B \to B$, a set S containing elements of type A and an element b of type B. It is inductively defined: if $S \neq \emptyset$, it returns $\texttt{fold}(f, S \setminus s, f(s,b))$ (where $s \in S$). Otherwise, it returns b.

3 Decentralized Runtime Enforcement

In this section, we first define the decentralized runtime problem, stating our assumptions. Then, we define the requirements on decentralized enforcers.

Problem statement and assumptions. Let $i_k \in \Sigma_k^*$ (with $k \in [1 \dots n]$) be the local trace of each component C_k and $i \in \Sigma^*$ the global trace obtained from the union of the local traces. For $k \in [1 \dots n]$, local trace i_k is input to enforcer M_k. Similarly, let $o_k \in \Sigma_k^*$ be the local output trace of each enforcer and $o \in \Sigma^*$ the global output trace obtained from the union of the local outputs.

Our assumptions on the system are as follows:

- The formula formalizing the specification is not equivalent to $\bot$.
- The system cannot emit a new event until the previous one has been treated by the enforcer.
- An enforcer M_k can only read and modify the atomic propositions in AP_k.
- All enforcers are capable to communicate with one another by exchanging messages.
- All exchanged messages are delivered reliably, in order, and with no alteration.
- Enforcers are not malicious, i.e., they do not exchange wrong information.

Intuitively, every time a new event σ is emitted by the system, the enforcers compute the set of events that respect the specification using their local observations of σ. They will then choose to emit one of these events and modify their local observations accordingly. We denote this event by $E(\sigma)$. At time

t, we have $E(i) = E(i(1)) \cdot E(i(2)) \cdots E(i(t)) = o$ and $E(i_k) = E(i_k(1)) \cdot E(i_k(2)) \cdots E(i_k(t)) = o_k$. Let φ be the formula representing the specification to be enforced of the system and AP the set of atomic propositions present in φ. We want to obtain online decentralized enforcers so that if $\sigma \in bad(\varphi)$, then $E(\sigma) \notin bad(\varphi)$.

Example 1 (Running example) *We will illustrate each part of the subsequent enforcement algorithms on formula $\phi = \neg(\mathbf{G}a \vee \mathbf{F}b)$. We consider an example system with two components C_1 and C_2. We use two enforcers M_1 and M_2 with $AP_1 = \{a\}$ and $AP_2 = \{b\}$. The initial event emitted by the system is $\sigma = \{a\}$ and the enforcer M_1 is doing the initialization (this is an arbitrary choice).*

For the remainder of this paper, unless specified otherwise, $\sigma \in \Sigma$ represents the global event emitted by the system, $E(\sigma) \in \Sigma$ denotes the event outputted by the enforcers and $\varphi \in LTL$ represents the formula to be enforced. At timestamp 1, φ is equal to the specification formula φ_{init} and then, at timestamp $t+1$, φ is equal to φ^{t+1}, the formula obtained at the end of the t-th timestamp.

Requirements on enforcers. We define the requirements on a decentralized enforcer E.

Definition 2 (Soundness) $\forall i \in \Sigma^*,\ E(i) \notin bad(\varphi)$.

An enforcer is *sound* if its output is not a bad prefix of the specification.

Definition 3 (Transparency) $\forall i \in \Sigma^*, \forall \sigma \in \Sigma,\ E(i) \cdot \sigma \notin bad(\varphi) \Rightarrow E(i \cdot \sigma) = E(i) \cdot \sigma$.

An enforcer is *transparent* if its input is modified only when it leads to a violation of the specification. We also define the notion of *optimality*.

Definition 4 (Optimality)

$$\begin{aligned}\forall i \in \Sigma^*,\ \forall \sigma \in \Sigma,\ \exists \sigma' \in \Sigma,\\ E(i \cdot \sigma) = E(i) \cdot \sigma'\\ \wedge\ \forall \sigma'' \in \Sigma,\ \texttt{distance}(\sigma, \sigma'', AP) < \texttt{distance}(\sigma, \sigma', AP) \Longrightarrow E(i) \cdot \sigma'' \in bad(\varphi).\end{aligned}$$

An enforcer is *optimal* if it outputs the closest event to the input that respects the property.

4 Normalizing LTL Formulas

We transform the formula into its Temporal Disjunctive Normal Form (TDNF) to get a disjunction of monomials. For this, we start by separating present and future obligations in the formula (Sec. 4.1) and then use an algorithm to transform the result of the previous step into its DNF (Sec. 4.2). These two operations applied one after the other on the input formula yield the TDNF, in which each monomial represents a logical model of the formula and is the conjunction of two sub-formulas: a state formula (i.e. the present obligation) and a conjunction of temporal operators (i.e. the future obligation).

4.1 Separating Present and Future Obligations

The enforcers determine whether or not there is a violation of the set of properties by evaluating the corresponding formula with σ. To achieve this, we use the expansion laws (as defined in [27]) to separate what σ needs to satisfy in the current timestamp, i.e. the present obligations, from what needs to be satisfied in the future, i.e. the future obligations. For example, to evaluate $\mathbf{G}a$, we first need to rewrite it as $a \wedge \mathbf{X}(\mathbf{G}a)$. We can see that, after rewriting, a has to hold on the current event σ and that $\mathbf{G}a$ has to hold in the future.

Definition 5 (Expansion function (`rwT`)) *Let $\varphi, \varphi_1, \varphi_2 \in LTL$. Function rwT: $LTL \to LTL$ is inductively defined as follows:*

$$
\begin{aligned}
rwT(p) &= p, \text{ for } p \in AP & rwT(\top) &= \top \\
rwT(\varphi_1 \vee \varphi_2) &= rwT(\varphi_1) \vee rwT(\varphi_2) & rwT(\bot) &= \bot \\
rwT(\varphi_1 \wedge \varphi_2) &= rwT(\varphi_1) \wedge rwT(\varphi_2) & rwT(\neg\varphi) &= \neg rwT(\varphi) \\
rwT(\varphi_1 \Rightarrow \varphi_2) &= rwT(\varphi_1) \Rightarrow rwT(\varphi_2) & rwT(\mathbf{X}\varphi) &= \mathbf{X}\varphi \\
rwT(\varphi_1 \mathbf{U} \varphi_2) &= rwT(\varphi_2) \vee (rwT(\varphi_1) \wedge \mathbf{X}(\varphi_1 \mathbf{U} \varphi_2)) & rwT(\mathbf{G}\varphi) &= rwT(\varphi) \wedge \mathbf{X}(\mathbf{G}\varphi) \\
rwT(\varphi_1 \mathbf{R} \varphi_2) &= rwT(\varphi_2) \wedge (rwT(\varphi_1) \vee \mathbf{X}(\varphi_1 \mathbf{R} \varphi_2)) & rwT(\mathbf{F}\varphi) &= rwT(\varphi) \vee \mathbf{X}(\mathbf{F}\varphi)
\end{aligned}
$$

Example 2 (`rwT`) *Recall that $\phi = \neg(\mathbf{G}a \vee \mathbf{F}b)$. We have $rwT(\phi) = \neg(a \wedge \mathbf{X}(\mathbf{G}a) \vee b \vee \mathbf{X}(\mathbf{F}b)) = \phi'$.*

Any formula outputted by function `rwT` is semantically equivalent to the input formula:

Property 1 $\forall \varphi \in LTL,\ \varphi \equiv rwT(\varphi)$.

Moreover, thanks to the expansion laws, the formulas produced by `rwT` satisfy the following syntactic property.

Property 2 *Let $\varphi \in LTL$. In the syntactic tree of $rwT(\varphi)$, any temporal operator different from* **X** *is below a* **X**.

4.2 Transforming to Temporal Disjunctive Normal Form

A problem that arises now is knowing which formula has to be evaluated in the future based on the current observation. For example, with $a\mathbf{U}b$, the formula to evaluate in the future will be either $\top$ or $a\mathbf{U}b$ depending on the values of a and b in the current event.

Each monomial of a formula in DNF represents one of its model. The transformation to DNF gives, for each model, a formula of the form $\varphi_1 \wedge \mathbf{X}\varphi_2$ where φ_1 is a conjunction of literals and φ_2 is a conjunction of temporal operators, that is, φ_1 (resp. φ_2) represents the present obligation (resp. future obligation). Intuitively, if the present obligations of a monomial hold, then the corresponding future obligations should hold on the trace later on and should therefore be included in the formula that needs to be evaluated during the next timestamp.

In the following, we use a function `DNF` that transforms any formula in DNF. The details of each step of the transformation can be found in [19].

Example 3 (Transformation to DNF) *We transform ϕ' into its DNF:*

$$DNF(\phi') = \neg a \wedge \neg b \wedge \mathbf{X}(\mathbf{G}\neg b) \vee \mathbf{X}(\mathbf{F}\neg a) \wedge \neg b \wedge \mathbf{X}(\mathbf{G}\neg b).$$

We denote $DNF(\phi')$ by ϕ_{DNF}.

Any formula outputted by function `DNF` is semantically equivalent to the input formula:

Property 3 $\forall \varphi \in LTL,\ \varphi \equiv DNF(\varphi)$.

Moreover, the transformation to DNF ensures the following syntactic property:

Property 4 *Let φ be an LTL formula that has been rewritten by rwT. In the syntactic tree of $DNF(\varphi)$, any temporal operator different from* **X** *is below a* **X**.

Corollary 1 *A formula rewritten by* `rwT` *and then transformed into its DNF with* `DNF` *is in TDNF.*

Definition 6 (Present obligation) *LTL_{p} is the set of formulas representing the present obligations. It is defined by the following grammar (where $p \in AP$): $\varphi_p \in LTL_{\text{p}} ::= \neg p \mid p \mid \top \mid \bot \mid \varphi_p \wedge \varphi_p$.*

Definition 7 (Future obligation) *LTL_{f} is the set of formulas representing the future obligations. It is defined by the following grammar (where $\varphi \in LTL$): $\varphi_f \in LTL_{\text{f}} := \mathbf{X}\varphi \mid \varphi_f \wedge \varphi_f$.*

5 Temporal Enforcement Encoding

Normalization (Sec. 4) provides a clear separation between present and future and ensures that the specification formula is in TDNF. We define in Sec. 5.1 an encoding that associates each monomial of the formula with a pair (*present*, *future*) where *present* (resp. *future*) is a formula that represents the present obligations (resp. future obligations). Then, we define the partial function representing the state of an enforcer in Sec. 5.2.

5.1 Encoding Present and Future: Temporal Obligation Pairs (TOP)

The enforcers are now able to rewrite and evaluate the present obligations to determine whether or not σ leads to a violation of the specification. To represent the whole formula, we generate a set containing a pair for each monomial of the formula. We refer to these pairs as *temporal obligation pairs* (TOP). As the set represents a disjunction of monomials, there is a violation iff the present obligations of every pair in the set evaluate to $\bot$.

This separation allows the enforcers to only work on the present obligations. This is useful because, for example, at timestamp t, the next formula to be enforced φ^{t+1} contains some of the future obligations of φ^t but not necessarily all of them. Let *present* $\wedge$ *future* be a monomial in φ^t. If $E(\sigma) \vDash$ *present*, then *future* is included in φ^{t+1}. Evaluating future obligations is useless if they are not included in the next formula to be enforced. To illustrate this, consider formula $a\mathbf{U}b$, we have $\texttt{rwT}(a\mathbf{U}b) = b \vee (a \wedge \mathbf{X}(a\mathbf{U}b))$. We can see here that if $b \in \sigma$ (b holds), then the trace should satisfy $\top$ in the future and that, if only a is true, then the trace should satisfy $a\mathbf{U}b$ in the future. Therefore, we associate $a\mathbf{U}b$ with $\{(b, \top), (a, a\mathbf{U}b)\}$. Furthermore, splitting the formula to be enforced into smaller formulas reduces the cost of the simplification because we do not need to rewrite the future obligations in the current timestamp as stated above.

We define function `encode` which transforms an *LTL* formula into a set of pairs (*present*, *future*). We assume that the input formula has already been rewritten by the expansion function and then transformed into its TDNF. We know from Property 4 that any temporal operators is below a $\mathbf{X}$. Therefore, we do not need to define the following function for temporal operators as there cannot be any in present obligations.

Definition 8 (Temporal obligation pair) *Let $\varphi, \varphi_1, \varphi_2 \in LTL$. Function $\texttt{encode} : LTL \to 2^{LTL_{\text{p}} \times LTL_{\text{f}}}$ is defined as follows:*

$$\texttt{encode}(\varphi_1 \vee \varphi_2) = \texttt{encode}(\varphi_1) \cup \texttt{encode}(\varphi_2) \qquad \texttt{encode}(\varphi_1 \wedge \varphi_2) = \{(p_{\varphi_1} \wedge p_{\varphi_2}, f_{\varphi_1} \wedge f_{\varphi_2})\} \textit{ where}$$
$$\texttt{encode}(\neg p) = \{(\neg p, \top)\}, \textit{with } p \in AP \qquad \{(p_{\varphi_1}, f_{\varphi_1})\} = \texttt{encode}(\varphi_1) \textit{ and}$$
$$\texttt{encode}(p) = \{(p, \top)\}, \textit{with } p \in AP \qquad \{(p_{\varphi_2}, f_{\varphi_2})\} = \texttt{encode}(\varphi_2)$$
$$\texttt{encode}(\mathbf{X}\varphi) = \{(\top, \varphi)\}$$

Example 4 (Temporal obligation pair) *Let* $\phi_1 = \neg a \wedge \neg b \wedge \mathbf{X}(\mathbf{G}\neg b)$ *and* $\phi_2 = \mathbf{X}(\mathbf{F}\neg a) \wedge \neg b \wedge \mathbf{X}(\mathbf{G}\neg b)$*, we have* $\texttt{encode}(\phi_1) = \{(\neg a \wedge \neg b,\ \mathbf{G}\neg b)\}$ *and* $\texttt{encode}(\phi_2) = \{(\neg b,\ \mathbf{F}\neg a \wedge \mathbf{G}\neg b)\}$*. As* $\phi_{DNF} = \phi_1 \vee \phi_2$*, we have* $\texttt{encode}(\phi_{DNF}) = \{(\neg a \wedge \neg b,\ \mathbf{G}\neg b), (\neg b, \mathbf{F}\neg a \wedge \mathbf{G}\neg b)\}$*. We denote by* ϕ_{TOP} *the result of* $\texttt{encode}(\phi_{DNF})$*.*

Property 5 *Let* $S \in 2^{LTL_{\text{p}} \times LTL_{\text{f}}}$*. We have:* $\forall \varphi \in LTL,\ \texttt{encode}(\varphi) = S \Longrightarrow \varphi \equiv \bigvee_{(p,f)\in S} p \wedge f$*.*

Using the semantics of LTL, we obtain the following corollary.

Corollary 2 *Let* $\sigma \in \Sigma^*$*,* $\sigma \notin bad(\varphi)$ *iff* $\exists (p,f) \in \texttt{encode}(\varphi), \sigma \notin bad(p) \wedge (sigma \notin bad(f)$*.*

5.2 Temporal Correction Log (TCL, Enforcer State)

To enforce the specification, we explore the correction events σ' s.t. $\sigma' \notin bad(\varphi)$. For this, we define the Temporal Correction Log (TCL) which serves as a state of the enforcer, encoding the status of the exploration. The TCL is a function that associates events of the alphabet with a pair containing a set of temporal obligation pairs and a natural number. We denote the set of all possible TCL by $\texttt{TCL}$ and an object from this set by $\texttt{tcl}$. When the event observed by the enforcers is σ and $\texttt{tcl}(\sigma') = (S,n)$, it means that the LTL formula that would need to be satisfied if the enforcers choose to produce event σ' as output is encoded by the set of TOP S and that the distance between σ and σ' is n. A TCL is built incrementally: each enforcer updates the values associated with the events using their local observations. It is a partial function $\texttt{tcl} : \Sigma \to 2^{LTL_p \times LTL_f} \times \mathbb{N}$. Initially, the state of the enforcers is initialized to $[\emptyset \mapsto (\texttt{encode}(\texttt{DNF}(\texttt{rwT}(\varphi))), 0)]$.

6 Evaluating the Formula

After initializing their state, the enforcers have to evaluate the formula to be able to choose the output event. To achieve this, they update their state using their local observations and send it to other enforcers to gather information on the global event. In Sec. 6.1, we define how the state of an enforcer is updated using its local observation and we then give in Sec. 6.2 a function to reduce the size of the state of an enforcer as well as defining the communication between the enforcers.

6.1 Updating the Temporal Correction Log

When an enforcer receives the state of another one (i.e. a TCL), it updates its domain by adding its local observations. Each new event is associated with an updated pair and the old events are removed from the domain. The set of TOP is updated by rewriting the present obligations and the distance metric is updated using the new local observation.

We define the local function used by each enforcer for the rewriting of the present obligations using a local observation $\sigma'' \in \Sigma_i$. This function uses the set of local atomic propositions to differentiate an atomic proposition that does not hold on σ'' from one that has not been observed yet.

Definition 9 (Rewriting of the present obligations) *Let* $\varphi, \varphi_1, \varphi_2 \in LTL_{\text{p}}$ *and* $\sigma'' \in 2^{AP_i}$*. On enforcer* M_i*, function* rw_i*:* $LTL_{\text{p}} \times \Sigma_i \to LTL_{\text{p}}$ *is defined as:*

$$rw_i(p \in AP, \sigma'') = \begin{cases} \top & \text{if } p \in \sigma'' \\ \bot & \text{if } p \notin \sigma'' \wedge p \in AP_i \\ p & \text{otherwise} \end{cases} \qquad rw_i(\neg\varphi, \sigma'') = \neg rw_i(\varphi, \sigma'')$$

$$rw_i(\varphi_1 \vee \varphi_2, \sigma'') = rw_i(\varphi_1, \sigma'') \vee rw_i(\varphi_2, \sigma'') \qquad rw_i(\varphi_1 \wedge \varphi_2, \sigma'') = rw_i(\varphi_1, \sigma'') \wedge rw_i(\varphi_2, \sigma'')$$

Note: These cases are sufficient because we only use function $\texttt{rw}_i$ on present obligations. Therefore, there is no temporal operators (nor implications or equivalences thanks to function `DNF`).

We now build the set of atomic propositions that still need to be evaluated (the remaining atomic propositions in the formula). When this set is empty, every atomic proposition of the formula has been evaluated, which means the evaluation phase is over and we then need to decide about the event to emit. Therefore, we define function $\texttt{apr} : \texttt{TCL} \to 2^{AP}$, which builds this set using `apFormula`. More precisely, `apr` yields the union of the result of `apFormula` on the present obligations of every set of TOP in the codomain of the `tcl`, that is, $\texttt{apr(tcl)} = \texttt{fold}(\cup,\ \texttt{map}(\texttt{apFormula}, \{p \mid (p,f) \in \texttt{top}, (\texttt{top}, -) \in \mathit{codom}(\texttt{tcl})\}), \emptyset)$.

Additionally, to prevent useless rewritings with atomic propositions that are not in the formula, we update the `tcl` using each event from $2^{AP_i \cap \texttt{apr(tcl)}}$ instead of Σ_i. For example, in formula $\mathbf{F}a$, we have $\texttt{apr(tcl)} = \{a\}$. Now suppose an enforcer M with $AP_i = \{a,b\}$ has to update that formula. M can observe four events locally: $\emptyset, \{a\}, \{b\}$ and $\{a,b\}$. However, as b is not in the formula, $\emptyset$ and $\{b\}$ both yield the same result after rewriting (likewise with $\{a\}$ and $\{a,b\}$). Therefore, we can see that evaluating the formula using atomic propositions that are not present in it leads to "useless" rewriting because the result is the same with at least one other local observation. This also means that each enforcer only receives the TCL once during the evaluation as all the atomic propositions they can observe locally are replaced by either $\top$ or $\bot$. In the remainder of this paper, we denote $2^{AP_i \cap \texttt{apr(tcl)}}$ by Σ_i^r which represents, intuitively, the set of all local observations of enforcer M_i containing only atomic propositions that are present in the formula.

Example 5 (Rewriting of the present obligations) *Recall that* $\phi_{TOP} = \{(\neg a \wedge \neg b, \mathbf{G}\neg b), (\neg b, \mathbf{F}\neg a \wedge \mathbf{G}\neg b)\}$. *After initialization, we have dom(*`tcl`*) =* $\{\emptyset\}$ *and* `tcl`$(\emptyset) = (\phi_{TOP}, 0)$. *Here, the present obligation of the first monomial contains a and b, the ones from the second monomial only contains b. Therefore,* $\texttt{apr(tcl)} = \texttt{apFormula}(\neg a \wedge \neg b) \cup \texttt{apFormula}(\neg b) = \{a,b\}$. *We have* $AP_1 = \{a\}$ *so* $\Sigma_1 = \{\emptyset, \{a\}\} = \Sigma_1^r$. *Evaluating the present obligations using the local observations of* M_1 *yields the following formulas:*

$$\texttt{rw}_i(\neg a \wedge \neg b, \emptyset) = \neg b \qquad \texttt{rw}_i(\neg b, \emptyset) = \neg b \qquad \texttt{rw}_i(\neg a \wedge \neg b, \{a\}) = \bot \qquad \texttt{rw}_i(\neg b, \{a\}) = \neg b$$

Algorithm 1 Update of the current enforcer state: `updateTCL(tcl, σ, apr(tcl))`

1: **for** each $\sigma' \in \mathit{dom}(\texttt{tcl})$ **do**
2: Let $(pf, n) = \texttt{tcl}(\sigma')$
3: $\mathit{dom}(\texttt{tcl}) = \mathit{dom}(\texttt{tcl}) \setminus \sigma'$
4: **for** each $\sigma'' \in \Sigma_i^r$ **do**
5: Let $pf' = \texttt{map}(\lambda(p,-).\texttt{rw}_i(p,\sigma''),\ pf)$
6: $\texttt{tcl} = \texttt{tcl}[\sigma'' \cup \sigma' \to (pf',\ n + \texttt{distance}(\sigma'', \sigma, AP_i \cap \texttt{apr(tcl)}))]$
7: **end for**
8: **end for**

We now define function $\texttt{updateTCL} : \texttt{TCL} \times \Sigma_i \times AP \to \texttt{TCL}$ used by the enforcers to update their state. The function takes three arguments: the state of enforcer to update, its local observation of the global event and the set of atomic propositions that have not been evaluated yet in any present obligation. It is updated using every $\sigma'' \in \Sigma_i^r$. The domain becomes the set $\sigma' \cup \sigma''$ (with $\sigma' \in \mathit{dom}(\texttt{tcl})$, an event of the "old" domain). Let pr be some present obligation. pr is rewritten as follows: $pr = \texttt{rw}_i(pr, \sigma'')$. The distance is updated: it is increased by 1 for each atomic proposition $p \in \sigma''$ that has a different truth value compared to σ, that is, if the metric was equal to n before the update, it becomes $n + \texttt{distance}(\sigma, \sigma'', AP_i \cap \texttt{apr(tcl)}))$. The definition of `updateTCL` is given in Algorithm 1.

Example 6 (`updateTCL`) *Recall that* $AP_1 = \{a\}$, $\texttt{apr}(\texttt{tcl}) = \{a,b\}$ *and* $\sigma = \{a\}$. *Before* M_1 *updates* `tcl`, *we have* $\texttt{tcl}(\emptyset) = (\phi_{TOP}, 0) = (\{(\neg a \wedge \neg b,\ \mathbf{G}\neg b), (\neg b, \mathbf{F}\neg a \wedge \mathbf{G}\neg b)\},\ 0)$. *After updating* `tcl` *with* `updateTCL`, *we have:*

$$\texttt{tcl}(\emptyset) = (\{(\neg b, \mathbf{G}\neg b), (\neg b, \mathbf{F}\neg a \wedge \mathbf{G}\neg b)\},\ 1) \quad \texttt{tcl}(\{a\}) = (\{(\bot, \mathbf{G}\neg b), (\neg b, \mathbf{F}\neg a \wedge \mathbf{G}\neg b)\},\ 0)$$

6.2 Reduction of the Obligations Set and Communication

An issue induced by Algorithm 1 is that the size of the domain of the `tcl` doubles for each atomic propositions in $AP_i \cap \texttt{apr}(\texttt{tcl})$. Therefore, we reduce its size by removing certain elements that become useless after rewriting. First, we reduce the size of the elements of the codomain (the images) by removing the pairs in which either the present or the future obligations have been evaluated to $\bot$, i.e. the monomials evaluated to $\bot$. Then, we remove from the domain the events that are associated with a pair containing an empty set of TOP as these events do not satisfy any present obligations. The second reduction guarantees that any event in the domain at the end of the evaluation does not lead to a violation of φ. Therefore, this also guarantees that adding the emitted event to the trace will not form a bad prefix of the formula.

Algorithm 2 Reduce the size of the domain of the TCL

1: **for** each $\sigma' \in dom(\texttt{tcl})$ **do**
2: Let $(pf, -) = \texttt{tcl}(\sigma')$
3: $pf' = \texttt{filter}((\lambda(p,f).(p \not\equiv \bot \wedge f \not\equiv \bot)),\ pf)$
4: **if** $pf' == \emptyset$ **then**
5: $dom(\texttt{tcl}) = dom(\texttt{tcl}) \setminus \sigma'$
6: **else**
7: $\texttt{tcl} = \texttt{tcl}[\sigma' \rightarrow (pf',\ n)]$
8: **end if**
9: **end for**

For this, we define `reduce`: `TCL` $\rightarrow$ `TCL` which implements the two aforementioned reductions in Algorithm 2.

Example 7 (`reduce`) *In example 5, we got* $\texttt{tcl}(\{a\}) = (\{(\bot, \mathbf{G}\neg b), (\neg b, \mathbf{F}\neg a \wedge \mathbf{G}\neg b)\},\ 0)$. *The set of obligations associated with this event contains a monomial in which the present obligations have been evaluated to* $\bot$. *Therefore,* $\texttt{tcl}(\{a\}) = (\{(\neg b, \mathbf{F}\neg a \wedge \mathbf{G}\neg b)\},\ 0)$.

After this reduction, if there still is at least one atomic proposition to evaluate, then the current enforcer communicates its state to another enforcer that can update it, i.e. an enforcer that can locally observe one of the remaining atomic propositions in any of the present obligations. If there are multiple enforcers that can rewrite the formula, the one with the smallest index is chosen. If the formula has been evaluated in its entirety, the decision rule can be applied to choose the event to emit.

Example 8 (Communication) *We can see in the previous example that b has not been evaluated yet so the* `tcl` *is sent to a enforcer that can observe this atomic proposition:* M_2, *in that case.*

7 Enforcement using a Decision based on Global Exploration

We now define the decision rule applied by the last enforcer in Sec. 7.1 and the algorithm itself in Sec. 7.2. We state some properties of the algorithm in Sec. 7.3.

7.1 Decision Rule

The decision rule is used by the enforcers to determine the emitted event. Let $\texttt{tcl}_f$ be the partial function representing the final state of the enforcer. Once an event has been chosen, each enforcer will modify its

local observations accordingly (if needed). The last enforcer that updates the `tcl` needs to send its state to all the other enforcers so that they all apply the decision rule and choose an event in parallel.

To respect *transparency*, we simply choose the event that has the least number of changes compared to σ. We know (from section 6.2), that the events σ' so that $u \cdot \sigma' \in bad(\varphi)$ (with $u \in \Sigma^*$ the trace up until the current timestamp and $\varphi \in LTL$ the property to enforce) have been removed from the domain of $\texttt{tcl}_f$. Therefore, if $u \cdot \sigma \notin bad(\varphi)$, then σ is the only element of the domain with its distance equal to 0 so σ will be emitted. If $u \cdot \sigma \in bad(\varphi)$, it is possible to have multiple events with the same distance. Let $\texttt{tcl}_{\text{candidates}}$ be the set of events with the least number of changes from σ: $\texttt{tcl}_{\text{candidates}} = \{\sigma' \in \texttt{tcl}_f \mid \texttt{tcl}_f(\sigma') = (-, n_{\min})\}$ so that $n_{\min} = min(\{n \mid (-,n) \in codom(\texttt{tcl}_f)\})$. If σ' is chosen, then the local event emitted by each enforcer M_i is $\sigma' \cap AP_i$.

If we choose to emit event σ', then φ^{t+1} is the disjunction of the future obligations associated with the emitted event and we have $\varphi^{t+1} = \texttt{fold}(\ (\lambda x,(-,f).(x \vee f)),\ pf,\ \bot)$ with $(pf,-) = \texttt{tcl}_f(\sigma')$. Therefore, if $(\top,\top)$ is included in the set of TOP of any events in $\texttt{TCL}_{\text{candidates}}$, then $\varphi^{t+1} = \top$. In this particular situation, we can stop the enforcers (any event is a good prefix of $\top$) so, if there exists $\sigma' \in \texttt{tcl}_{\text{candidates}}$ so that $\texttt{TCL}(\sigma') = (pf,-) \wedge (\top,\top) \in pf$ then we reduce $\texttt{TCL}_{\text{candidates}}$ so that it only contains these σ'. Finally, if we still have $\#\texttt{TCL}_{\text{candidates}} > 1$, we choose the event to emit from this set arbitrarily (but deterministically). Every enforcer applies the decision rule but the choice of the verdict is deterministic so they will all choose the same one. This implies that φ^{t+1} is the same for all of them which means that they all know the next formula to enforce without any additional communication.

Example 9 (Decision rule) *After the update of M_2, we have:*

$$\texttt{tcl}(\emptyset) = (\{(\top,\top),(\top,\mathbf{F}\neg a \wedge \mathbf{G}\neg b)\},\ 1) \qquad \texttt{tcl}(\{a\}) = (\{(\top,\mathbf{F}\neg a \wedge \mathbf{G}\neg b)\},\ 0)$$

The events $\{a,b\}$ and $\{b\}$ have been removed as they lead to a violation of ϕ ($\mathbf{G}\neg b$ evaluates to $\bot$ if $b = \top$). We apply the decision rule to choose the event to emit. Here, σ is chosen because it does not lead to a violation so we do not need to modify it. The local event emitted by M_1 (resp. M_2) is $\sigma'' = \{a\} \in \Sigma_1$ (resp. $\sigma'' = \emptyset \in \Sigma_2$). The formula that needs to be monitored during the next timestamp is $\phi^{t+1} = \mathbf{F}\neg a \wedge \mathbf{G}\neg b$.

7.2 Enforcement Algorithm

Let $\mathscr{M} = \{M_1,\ldots,M_n\}$ be the set of enforcers. Algorithm 3 is a local algorithm run by each enforcer. Enforcer M_1 is chosen arbitrarily to be the one initializing the evaluation. M_1 also updates its state immediately if it is able to, i.e. if the formula contains an atomic propositions in AP_1. All the other enforcers start by waiting to receive a `tcl`. Each enforcer M_j takes its local observation of the global event $\sigma \in \Sigma$ emitted by the system as input. Once the computation is over, they output their local observations of the event $\sigma' \in \Sigma$, that is different from σ iff $u \cdot \sigma \in bad(\varphi)$, where u is the trace up until the current timestamp t, $u \in \Sigma^*$. The enforcer also builds the formula φ^{t+1} based on the emitted event.

7.3 Properties

The number of messages sent is bounded: in the worst case scenario, that is, if the formula contains at least one local observation of each enforcer, the `tcl` is sent to every enforcer during the evaluation and once more at the end so that every enforcer can apply the decision rule. Therefore, if we denote by δ_{init} the time needed to initialize the state of the enforcer (Algorithm 2), by δ_{update} the time needed for one

Algorithm 3 Enforcement on M_j using a decision based on global exploration

1: **Initialization:** If $j == 1$, $\texttt{tcl}$ is initialized to $[\emptyset \mapsto (\texttt{encode}(\texttt{DNF}(\texttt{rwT}(\varphi^t))), 0)]$.
Otherwise, wait until a $\texttt{tcl}$ is received from another enforcer. If $\texttt{apr}(\texttt{tcl}) = \emptyset$ in the received $\texttt{tcl}$, jump to step 5.
2: **Evaluation:** Compute $\texttt{apr}(\texttt{tcl})$, the set of atomic propositions that have not been evaluated yet. Update $\texttt{tcl}$ using $\texttt{updateTCL}(\texttt{tcl}, \sigma, AP_j \cap \texttt{apr}(\texttt{tcl}))$.
3: **Reduction:** Reduce the domain of $\texttt{tcl}$ by removing the monomials that evaluate to $\bot$ and the bad prefixes of φ^t using $\texttt{reduce}(\texttt{tcl})$.
4: **Communication:** If $\texttt{apr}(\texttt{tcl}) \neq \emptyset$, let $\mathscr{M}' \subseteq \mathscr{M}$ be the set of enforcers M_k so that $AP_k \cap \texttt{apr}(\texttt{tcl}) \neq \emptyset \wedge j \neq k$. $\texttt{tcl}$ is sent to enforcer $M_{k_{\min}}$ with $k_{\min} = min\{k \mid M_k \in \mathscr{M}'\}$. Wait until a $\texttt{tcl}$ is received from another enforcer.
Otherwise, send $\texttt{tcl}$ to all the other enforcers so that they can apply the decision rule.
5: **Decision:** Let $\texttt{tcl}_f$ be the final state of $\texttt{tcl}$. Apply the decision rule to choose the event σ' to emit and set φ^{t+1} to $\texttt{fold}(\ (\lambda x, (-, f).(x \vee f)),\ pf,\ \bot)$, with $(pf, -) = \texttt{tcl}_f(\sigma')$.

update of the $\texttt{tcl}$ (update with each element of $2^{AP_i \cap \texttt{apr}(\texttt{tcl})}$) and by δ_{decision} the time it takes to apply the decision rule. The delay between two events emitted by the system is, at worst, $\#\mathscr{M} \times \delta_{\text{update}} + \delta_{\text{init}} + \delta_{\text{decision}}$ (every enforcer executes the decision rule at the same time). The size of the messages is also bounded: as we rewrite the obligations using every possible events in Σ, the size of the domain doubles for each observed atomic proposition. Therefore, denoting by n_{AP} the number of atomic propositions in the formula, the size of the domain is bounded by $2^{n_{AP}}$ elements and the size of the last message sent can be at most $2^{n_{AP}-1}$. At worst, the domain contains $2^{\#AP}$ elements at the end. It is worth noting that the transformation to TDNF is costly (exponential in the size of the formula) and that, in the worst case scenario, if $\texttt{reduce}$ does not remove elements from the domain, its size grows exponentially as well, which means that, in turn, δ_{update} gets longer every timestamp. δ_{decision} is negligible compared to the other terms.

Lemma 1 *Let φ^t be the property to enforce at timestamp t and φ^{t+1} the formula built at the end of the algorithm (that will be monitored during timestamp $t+1$). If $\varphi^t \not\equiv \bot$, then $\varphi^{t+1} \not\equiv \bot$.*

Since the initial property is assumed to be not equivalent to $\bot$, Lemma 1 implies that Algorithm 3 cannot produce a formula that is equivalent to $\bot$.

Property 6 *By using Algorithm 3 as a local enforcer on each component, we obtain a sound, transparent, and optimal enforcer (as described in Sec. 3).*

8 Enforcement using a Decision based on Local Exploration

In this section, we define another enforcement algorithm where each enforcer takes a local decision before sending its state to the next enforcer: instead of sending the whole $\texttt{tcl}$, the current enforcer only sends a single entry with its image. This also means that the enforcers do not need to make a decision at the end of the enforcement round: the local emitted event corresponds to this local decision. The receiving enforcer then initializes its state using the received event and applies the updates to it. The point of this approach is to prevent the exponential growth of the domain of the $\texttt{tcl}$. The local decision rule is defined in Sec. 8.1 and the algorithm in Sec. 8.2. We compare the two versions in Sec. 8.3 and give the properties of the second version in Sec. 8.4. A complete example is given in Sec. 8.5.

8.1 Local Decision Rule

To allow the enforcers to make a local decision, we redefine the decision rule. Let $\texttt{tcl}$ be the partial function representing the state of the enforcer after its update. $\texttt{apr}(\texttt{tcl})$ is recomputed after the evaluation. Two cases are possible:

- If $\texttt{apr}(\texttt{tcl}) = \emptyset$, the algorithm stops and the current enforcer decides which event to emit from a set $\texttt{tcl}_{\text{candidates}} = \{\sigma' \in \texttt{tcl}_f \mid \texttt{tcl}_f(\sigma') = (-, n_{min})\}$ so that $n_{min} = min(N)$ with $N = \{n \mid (-,n) \in codom(\texttt{tcl}_f)\}$.
  If there exists $\sigma' \in \texttt{tcl}_{\text{candidates}}$ so that $\texttt{tcl}(\sigma') = (pf, -) \wedge (\top, \top) \in pf$, then $\texttt{tcl}_{\text{candidates}} = \{\sigma' \in \texttt{tcl}_{\text{candidates}} \mid \texttt{tcl}(\sigma') = (pf, -) \wedge (\top, \top) \in pf\}$ for the same reasons as in the first algorithm: if $\varphi^{t+1} = \top$, we do not need to enforce the formula anymore as any event is a good prefix of $\top$.
  Finally, if there are multiple elements in $\texttt{tcl}_{\text{candidates}}$, one is chosen arbitrarily (and deterministically).
- If $\texttt{apr}(\texttt{tcl}) \neq \emptyset$, the current enforcer decides which element of the domain it will send to the next enforcer (with its image). The element is chosen from the set $\texttt{tcl}_c = \{(\sigma', (pf, n_{min})) \mid \sigma' \in dom(\texttt{tcl}) \wedge \texttt{tcl}(\sigma') = (pf, n_{min})\}$ so that $n_{min} = min(N)$ with $N = \{n \mid (-,n) \in codom(\texttt{tcl}_f)\}$, that is, it is chosen among the elements that have the smallest distance to σ.
  If there are several events that respect the property mentioned above, the set is reduced to the events that have the most models, i.e. $\texttt{tcl}_c = \{(-, (pf, -)) \in \texttt{tcl}_c \mid \nexists (-, (pf', -)) \in \texttt{tcl}_c, pf \neq pf' \wedge |pf'| > |pf|\}$ (the formulas that have the highest amount of monomials). If there are still multiple elements in $\texttt{tcl}_c$, one is chosen arbitrarily (and deterministically).

Just as in the first algorithm, if σ' is chosen, then M_i outputs $\sigma' \cap AP_i$. As only the last enforcer that applies the decision rule has sufficient information to determine φ^{t+1}, it needs to be sent to the other enforcers at the end of the evaluation.

8.2 Enforcement Algorithm

To obtain the enforcement algorithm in the case of decision based on local exploration, we update Algorithm 3 as follows.

In step 1, the (current) enforcer waits for a $\texttt{tcl}$ or the formula to enforce in the next timestamp. If it receives a formula, the execution of the algorithm for the current timestamp stops.

In step 4, the enforcer always applies the local decision rule to choose an event from the domain. It then removes all the other events from the domain and, if $\texttt{apr}(\texttt{tcl}) \neq \emptyset$, it sends $\texttt{tcl}$ to the next enforcer using the same criteria as in the first algorithm. Otherwise, it waits for the formula to enforce in the next timestamp instead of a $\texttt{tcl}$. When it receives the formula, the current timestamp stops (the enforcer does not go to step 5).

In step 5, the enforcer does not apply the decision rule as it already did it in the previous step. It only builds the formula to enforce at the next timestamp and sends it to all the other enforcers.

8.3 Comparison

First, let us notice that the algorithm with the local decision rule guarantees *soundness* and *transparency*, but not *optimality*. Indeed, as the algorithm always sends an event associated with a set of future obligations that has at least one solution, there is always a solution. Otherwise, this event would not be in the domain. This guarantees soundness. Transparency is guaranteed since the input event is removed from the domain only if it violates the property. However, as the set of all possible events over AP is not

entirely explored, the event emitted at the end might not be the best (w.r.t. the distance to σ). Hence, *optimality* is not guaranteed.

Regarding performance, despite requiring the transformation to DNF during the initialization, the algorithm presents a significant improvement regarding the size of the messages and of the domain (it prevents their exponential growth). Furthermore, the size of the domain is much smaller, it only contains one element before the update instead of, at worst, 2^{k-1} after k updates, which allows the enforcers to only do one update per element of $2^{AP_i \cap \texttt{apr(tcl)}}$ instead of $\#2^{AP_i \cap \texttt{apr(tcl)}} \times 2^{k-1}$ updates (one per local observation per element of the domain).

However, the algorithm has some drawbacks: the decision rule has to be applied by each enforcer, although this is not that significant considering it is negligible compared to the other operations. Moreover, this version does not guarantee *optimality*. It is worth noting that the first version of the algorithm is better than this one in one very specific scenario: if the domain of `tcl` is reduced to a single element with `reduce` at the end of every update, then this version is worse because it does not improve the size of the messages/domain but the enforcers still have to apply the local decision rule after every update instead of once in the first version. Even in this situation, the difference is not significant unless there is a large number of enforcers.

8.4 Properties

Just as in the first version, the number of messages sent is bounded: in the worst case, that is, if the formula contains at least one local observation of each enforcer, one message is sent to each enforcer. An additional message is sent to all the enforcers (except 1) at the end to communicate φ^{t+1}. Therefore, the delay between two events emitted by the system is, at worst, $(\delta_{\text{update}} + \delta_{\text{decision}}) \times \#\mathscr{M} + \delta_{\text{init}}$ with $\delta_{\text{init}}, \delta_{\text{update}}$ and δ_{decision} defined as in Sec. 7.3. As we only send a pair containing an element of the domain of `tcl` and its image, the size of the message is quite small. It is not completely constant because the size of the image may vary (although not by much).

8.5 Complete Example: Decentralized Traffic Lights

Table 1: Enforcing $\mathbf{G}((g_1 \wedge g_3 \wedge \neg(g_2 \vee g_4)) \vee (\neg(g_1 \vee g_3) \wedge g_2 \wedge g_4))$ given event $\sigma = \{g_1, g_2, g_3\}$.

Present/Future	$\texttt{rwT}(\varphi) = (g_1 \wedge g_3 \wedge \neg(g_2 \vee g_4) \vee \neg(g_1 \vee \neg g_3) \wedge g_2 \wedge g_4) \wedge \mathbf{X}\varphi = \varphi_{rwT}$
Transf. to TDNF	$\texttt{DNF}(\varphi_{rwT}) = (g_1 \wedge g_3 \wedge \neg g_2 \wedge \neg g_4 \wedge \mathbf{X}\varphi) \vee\ (\neg g_1 \wedge \neg g_3 \wedge g_2 \wedge g_4 \wedge \mathbf{X}\varphi) = \varphi_{TDNF}$
Initial `tcl`	$[\emptyset \mapsto \texttt{encode}(\varphi_{TDNF}) = (\{(g_1 \wedge g_3 \wedge \neg g_2 \wedge \neg g_4, \varphi), (\neg g_1 \wedge \neg g_3 \wedge g_2 \wedge g_4, \varphi)\},\ 0)]$
Evaluation (M_1)	$[\emptyset \mapsto (\{(\bot, \varphi), (\neg g_3 \wedge g_2 \wedge g_4, \varphi)\},\ 1), \{g_1\} \mapsto (\{(g_3 \wedge \neg g_2 \wedge \neg g_4, \varphi), (\bot, \varphi)\},\ 0)]$
Decision (M_1)	The entry corresponding to $\{g_1\}$ is chosen, others are removed.
Evaluation (M_2)	$[\{g_1\} \mapsto \{(g_3 \wedge \neg g_4, \varphi)\},\ 1), \{g_1, g_2\} \mapsto (\{(\bot, \varphi)\},\ 0)]$
Decision (M_2)	$\{g_1\}$ is the only event left in the domain. It is therefore chosen by default.
Evaluation (M_3)	$[\{g_1\} \mapsto (\{(\bot, \varphi)\},\ 2), \{g_1, g_3\} \mapsto (\{(\neg g_4, \varphi)\},\ 1)]$
Decision (M_3)	$\{g_1, g_3\}$ is the only event left in the domain. It is therefore chosen by default.
Evaluation (M_4)	$[\{g_1, g_3\} \mapsto (\{(\top, \varphi)\},\ 1), \{g_1, g_3, g_4\} \mapsto (\{(\bot, \varphi)\},\ 2)]$
Decision (M_4)	$\{g_1, g_3\}$ is the only event left in the domain. It is therefore chosen by default.
Next formula	$\varphi^2 = \varphi$

We consider a crossroad and its four traffic lights. We have $AP_i = \{g_i, y_i, r_i\}, i \in \{1,2,3,4\}$. Variable g_i (resp. y_i and r_i) indicates whether or not the green (resp. yellow and red) light of the i-th traffic light is on ($g_i = \top$ means it is on). Let φ be the property representing the following specification: *At any time, exactly two opposed traffic lights must be green at the same time*. We have: $\varphi = \mathbf{G}((g_1 \wedge g_3 \wedge \neg(g_2 \vee g_4)) \vee (\neg(g_1 \vee g_3) \wedge g_2 \wedge g_4))$. Let M_1, M_2, M_3 and M_4 be the four enforcers associated with each traffic light (one per traffic light). Let $\sigma \in \Sigma$ be the event emitted by the system, $\sigma = \{g_1, g_2, g_3\}$ which means that only the green light of the first three traffic lights is on. All the transformations are given in Table 1. A detailed description of this example on both enforcement algorithms is in [19]. In the table, shaded pairs are removed from the set through function `reduce`. Then, events associated to an empty set of TOP are removed from the domain.

9 Conclusions and Future work

Conclusions. This paper introduces the problem of *decentralized runtime enforcement* for systems without a global observation/control point. We give two decentralized enforcement algorithms for LTL formulas. Both algorithms guarantee *soundness* and *transparency*. The first also guarantees the *optimality* of the modifications done by the enforcer in terms of distance to the original event emitted by the system while the second comes with a drastically reduced cost (both in terms of time and space).

Future work. The natural extension of this work is its implementation to empirically evaluate it (on LTL specification patterns [8], for example) in terms of computational and communication costs. Then, we plan to integrate it into the THEMIS tool [10] which currently only supports verification. Our enforcement algorithms can also be extended in several ways. First, using LTL formula rewriting has a few drawbacks and in particular, rewriting renders the analysis of the runtime behavior of monitors hard to predict as it depends on the simplification function applied to LTL formulas after rewriting. Alternatively, we consider encoding the specification using automata, for example. Finally, we shall also consider timed properties as they are much more expressive. There are some approaches for runtime enforcement of timed properties (see [17] for an overview), but all are centralized.

Acknowledgment. Y. Falcone acknowledges the support from the H2020-ECSEL-2018-IA call – Grant Agreement number 826276 (CPS4EU), the European Union's Horizon 2020 research and innovation programme - Grant Agreement number 956123 (FOCETA), from the French ANR project ANR-20-CE39-0009 (SEVERITAS), from the Auvergne-Rhône-Alpes research project MOAP, and from LabEx PERSYVAL-Lab (ANR-11-LABX-0025-01) funded by the French program Investissement d'avenir.

References

[1] Ezio Bartocci & Yliès Falcone, editors (2018): *Lectures on Runtime Verification - Introductory and Advanced Topics*. *Lecture Notes in Computer Science* 10457, Springer, doi:10.1007/978-3-319-75632-5. Available at `https://doi.org/10.1007/978-3-319-75632-5`.

[2] David A. Basin, Vincent Jugé, Felix Klaedtke & Eugen Zalinescu (2013): *Enforceable Security Policies Revisited*. *ACM Trans. Inf. Syst. Secur.* 16(1), pp. 3:1–3:26, doi:10.1145/2487222.2487225. Available at `https://doi.org/10.1145/2487222.2487225`.

[3] Andreas Bauer & Yliès Falcone (2016): *Decentralised LTL monitoring*. *Formal Methods Syst. Des.* 48(1-2), pp. 46–93, doi:10.1007/s10703-016-0253-8. Available at https://doi.org/10.1007/s10703-016-0253-8.

[4] Andreas Bauer, Martin Leucker & Christian Schallhart (2011): *Runtime Verification for LTL and TLTL*. *ACM Trans. Softw. Eng. Methodol.* 20(4), pp. 14:1–14:64, doi:10.1145/2000799.2000800. Available at https://doi.org/10.1145/2000799.2000800.

[5] Borzoo Bonakdarpour, Pierre Fraigniaud, Sergio Rajsbaum & Corentin Travers (2016): *Challenges in Fault-Tolerant Distributed Runtime Verification*. In Tiziana Margaria & Bernhard Steffen, editors: *Leveraging Applications of Formal Methods, Verification and Validation: Discussion, Dissemination, Applications*, Springer International Publishing, Cham, pp. 363–370, doi:10.1007/978-3-319-47169-3_27. Available at https://doi.org/10.1007/978-3-319-47169-3_27.

[6] Christian Colombo & Yliès Falcone (2016): *Organising LTL monitors over distributed systems with a global clock*. *Formal Methods in System Design* 49(1), pp. 109–158, doi:10.1007/s10703-016-0251-x. Available at https://doi.org/10.1007/s10703-016-0251-x.

[7] Egor Dolzhenko, Jay Ligatti & Srikar Reddy (2015): *Modeling Runtime Enforcement with Mandatory Results Automata*. *Int. J. Inf. Secur.* 14(1), p. 47–60, doi:10.1007/s10207-014-0239-8. Available at https://doi.org/10.1007/s10207-014-0239-8.

[8] Matthew B. Dwyer, George S. Avrunin & James C. Corbett (1999): *Patterns in Property Specifications for Finite-State Verification*. In: *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, Association for Computing Machinery, New York, NY, USA, p. 411–420, doi:10.1145/302405.302672. Available at https://doi.org/10.1145/302405.302672.

[9] Antoine El-Hokayem & Yliès Falcone (2017): *Monitoring decentralized specifications*. In Tevfik Bultan & Koushik Sen, editors: *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017*, ACM, pp. 125–135, doi:10.1145/3092703.3092723. Available at https://doi.org/10.1145/3092703.3092723.

[10] Antoine El-Hokayem & Yliès Falcone (2017): *THEMIS: a tool for decentralized monitoring algorithms*. In Tevfik Bultan & Koushik Sen, editors: *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017*, ACM, pp. 372–375, doi:10.1145/3092703.3098224. Available at https://doi.org/10.1145/3092703.3098224.

[11] Antoine El-Hokayem & Yliès Falcone (2020): *On the Monitoring of Decentralized Specifications: Semantics, Properties, Analysis, and Simulation*. *ACM Trans. Softw. Eng. Methodol.* 29(1), pp. 1:1–1:57, doi:10.1145/3355181. Available at https://doi.org/10.1145/3355181.

[12] Yliès Falcone, Jean-Claude Fernandez & Laurent Mounier (2012): *What can you verify and enforce at runtime?* *Int. J. Softw. Tools Technol. Transf.* 14(3), pp. 349–382, doi:10.1007/s10009-011-0196-8. Available at https://doi.org/10.1007/s10009-011-0196-8.

[13] Yliès Falcone, Thierry Jéron, Hervé Marchand & Srinivas Pinisetty (2016): *Runtime enforcement of regular timed properties by suppressing and delaying events*. *Sci. Comput. Program.* 123, pp. 2–41, doi:10.1016/j.scico.2016.02.008. Available at https://doi.org/10.1016/j.scico.2016.02.008.

[14] Yliès Falcone, Srdan Krstic, Giles Reger & Dmitriy Traytel (2021): *A taxonomy for classifying runtime verification tools*. *Int. J. Softw. Tools Technol. Transf.* 23(2), pp. 255–284, doi:10.1007/s10009-021-00609-z. Available at https://doi.org/10.1007/s10009-021-00609-z.

[15] Yliès Falcone, Leonardo Mariani, Antoine Rollet & Saikat Saha (2018): *Runtime Failure Prevention and Reaction*. In Ezio Bartocci & Yliès Falcone, editors: *Lectures on Runtime Verification - Introductory and Advanced Topics*, *Lecture Notes in Computer Science* 10457, Springer, pp. 103–134, doi:10.1007/978-3-319-75632-5_4. Available at https://doi.org/10.1007/978-3-319-75632-5_4.

[16] Yliès Falcone, Laurent Mounier, Jean-Claude Fernandez & Jean-Luc Richier (2011): *Runtime enforcement monitors: composition, synthesis, and enforcement abilities*. *Formal Methods Syst. Des.*

38(3), pp. 223–262, doi:10.1007/s10703-011-0114-4. Available at https://doi.org/10.1007/s10703-011-0114-4.

[17] Yliès Falcone & Srinivas Pinisetty (2019): *On the Runtime Enforcement of Timed Properties*. In Bernd Finkbeiner & Leonardo Mariani, editors: *Runtime Verification - 19th International Conference, RV 2019, Porto, Portugal, October 8-11, 2019, Proceedings*, *Lecture Notes in Computer Science* 11757, Springer, pp. 48–69, doi:10.1007/978-3-030-32079-9_4. Available at https://doi.org/10.1007/978-3-030-32079-9_4.

[18] Adrian Francalanza, Jorge A. Pérez & César Sánchez (2018): *Runtime Verification for Decentralised and Distributed Systems*. In Ezio Bartocci & Yliès Falcone, editors: *Lectures on Runtime Verification - Introductory and Advanced Topics*, *Lecture Notes in Computer Science* 10457, Springer, pp. 176–210, doi:10.1007/978-3-319-75632-5_6. Available at https://doi.org/10.1007/978-3-319-75632-5_6.

[19] Florian Gallay & Yliès Falcone (2021): *Decentralized LTL Enforcement*. Available at www.ylies.fr.

[20] Sylvain Hallé, Raphaël Khoury, Quentin Betti, Antoine El-Hokayem & Yliès Falcone (2018): *Decentralized enforcement of document lifecycle constraints*. *Inf. Syst.* 74(Part), pp. 117–135, doi:10.1016/j.is.2017.08.002. Available at https://doi.org/10.1016/j.is.2017.08.002.

[21] Chi Hu, Wei Dong, Yonghui Yang, Hao Shi & Fei Deng (2020): *Decentralized runtime enforcement for robotic swarms*. *Frontiers Inf. Technol. Electron. Eng.* 21(11), pp. 1591–1606, doi:10.1631/FITEE.2000203. Available at https://doi.org/10.1631/FITEE.2000203.

[22] Raphaël Khoury & Sylvain Hallé (2015): *Runtime Enforcement with Partial Control*. In Joaquín García-Alfaro, Evangelos Kranakis & Guillaume Bonfante, editors: *Foundations and Practice of Security - 8th International Symposium, FPS 2015, Clermont-Ferrand, France, October 26-28, 2015, Revised Selected Papers*, *Lecture Notes in Computer Science* 9482, Springer, pp. 102–116, doi:10.1007/978-3-319-30303-1_-7. Available at https://doi.org/10.1007/978-3-319-30303-1_7.

[23] Bettina Könighofer, Julian Rudolf, Alexander Palmisano, Martin Tappler & Roderick Bloem (2021): *Online Shielding for Stochastic Systems*. In Aaron Dutle, Mariano M. Moscato, Laura Titolo, César A. Muñoz & Ivan Perez, editors: *NASA Formal Methods - 13th International Symposium, NFM 2021, Virtual Event, May 24-28, 2021, Proceedings*, *Lecture Notes in Computer Science* 12673, Springer, pp. 231–248, doi:10.1007/978-3-030-76384-8_15. Available at https://doi.org/10.1007/978-3-030-76384-8_15.

[24] Aravind Natarajan, Himanshu Chauhan, Neeraj Mittal & Vijay K. Garg (2017): *Efficient abstraction algorithms for predicate detection*. *Theoretical Computer Science* 688, pp. 24–48, doi:10.1016/j.tcs.2015.12.037. Available at https://doi.org/10.1016/j.tcs.2015.12.037. Distributed Computing and Networking.

[25] Vinit A. Ogale & Vijay K. Garg (2007): *Detecting Temporal Logic Predicates on Distributed Computations*. In Andrzej Pelc, editor: *Distributed Computing*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 420–434, doi:10.1007/978-3-540-75142-7_32. Available at https://doi.org/10.1007/978-3-540-75142-7_32.

[26] Srinivas Pinisetty, Yliès Falcone, Thierry Jéron, Hervé Marchand, Antoine Rollet & Omer Nguena-Timo (2014): *Runtime enforcement of timed properties revisited*. *Formal Methods Syst. Des.* 45(3), pp. 381–422, doi:10.1007/s10703-014-0215-y. Available at https://doi.org/10.1007/s10703-014-0215-y.

[27] Amir Pnueli (1977): *The temporal logic of programs*. In: *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pp. 46–57, doi:10.1109/SFCS.1977.32. Available at https://ieeexplore.ieee.org/document/4567924.

[28] Matthieu Renard, Yliès Falcone, Antoine Rollet, Srinivas Pinisetty, Thierry Jéron & Hervé Marchand (2015): *Enforcement of (Timed) Properties with Uncontrollable Events*. In Martin Leucker, Camilo Rueda & Frank D. Valencia, editors: *Theoretical Aspects of Computing - ICTAC 2015 - 12th International Colloquium Cali, Colombia, October 29-31, 2015, Proceedings*, *Lecture Notes in Computer Science*

9399, Springer, pp. 542–560, doi:10.1007/978-3-319-25150-9_31. Available at https://doi.org/10.1007/978-3-319-25150-9_31.

[29] Grigore Rosu & Klaus Havelund (2005): *Rewriting-Based Techniques for Runtime Verification*. *Autom. Softw. Eng.* 12(2), pp. 151–197, doi:10.1007/s10515-005-6205-y. Available at https://doi.org/10.1007/s10515-005-6205-y.

[30] Prasanna Thati & Grigore Roşu (2005): *Monitoring Algorithms for Metric Temporal Logic Specifications*. *Electronic Notes in Theoretical Computer Science* 113, pp. 145–162, doi:10.1016/j.entcs.2004.01.029. Available at https://doi.org/10.1016/j.entcs.2004.01.029. Proceedings of the Fourth Workshop on Runtime Verification (RV 2004).