Incorporating Monitors in Reactive Synthesis without Paying the Price

Shaun Azzopardi[0000−0002−2165−3698], Nir Piterman[0000−0002−8242−5357], and Gerardo Schneider[0000−0003−0629−6853]

University of Gothenburg, Gothenburg, Sweden
{name.surname}@gu.se

Abstract. Temporal synthesis attempts to construct reactive programs that satisfy a given declarative (LTL) formula. Practitioners have found it challenging to work exclusively with declarative specifications, and have found languages that combine modelling with declarative specifications more useful. Synthesised controllers may also need to work with pre-existing or manually constructed programs. In this paper we explore an approach that combines synthesis of declarative specifications in the presence of an existing behaviour model as a monitor, with the benefit of not having to reason about the state space of the monitor. We suggest a formal language with automata monitors as non-repeating and repeating triggers for LTL formulas. We use symbolic automata with memory as triggers, resulting in a strictly more expressive and succinct language than existing regular expression triggers. We give a compositional synthesis procedure for this language, where reasoning about the monitor state space is minimal. To show the advantages of our approach we apply it to specifications requiring counting and constraints over arbitrarily long sequence of events, where we can also see the power of parametrisation, easily handled in our approach. We provide a tool to construct controllers (in the form of symbolic automata) for our language.

Keywords: synthesis · temporal logic · symbolic automata · monitoring

1 Introduction

Synthesis of programs from declarative specifications is an attractive prospect. Although thought prohibitive due to the theoretical hardness of LTL synthesis, recent improvements have made it a more reasonable endeavour, e.g. the identification of GR(1) [25], for which synthesis is easier, and development of tools such as Strix [21, 23] whose decomposition method allows for practical synthesis of full LTL. Limitations remain in the context of LTL, due to the inherent hardness of the problem. Beyond LTL there are also directions where the practicality of synthesis is not clear.

In addition to these algorithmic challenges, there are additional methodological challenges. Practitioners have identified that it is sometimes very challenging to write declarative specifications, and suggested to use additional modelling

[14, 22]. Furthermore, synthesised parts need to work alongside pre-existing or manually constructed parts (cf. [20]). This, however, further exacerbates the algorithmic challenge as the state-space of the additional parts needs to be reasoned about by the synthesis algorithm.

We argue that modelling could also be a practical way of dealing with some of the algorithmic challenges and advocate a partial use of synthesis, leaving parts that are impractical for synthesis to be manually modelled. This leaves the question of how to combine the two parts.

We suggest to compose automata with synthesised controllers by transfer of control rather than co-operation. We define a specification language with repeating and non-repeating *trigger* properties (cf. [2, 18]). Triggers are defined as environment observing automata/monitors, which transfer control to LTL formulas. Both aspects – control transfer and triggers – are familiar to practitioners and would be easy to use: control transfer is natural for software; and triggers are heavily used in industrial verification languages (cf. [2]).

We aim at triggers that are rich, succinct and easy to write. Thus, we use monitors extracted from symbolic executable automata inspired by DATEs [8]. Expressiveness of automata is increased by having variables that are updated by guarded transitions, which means that auomata can be infinite-state (but the benefits remain if they are restricted to finite-state). This choice of monitors allows to push multiple other interesting concerns that are difficult for LTL synthesis to the monitor side. Experience of using such monitors in the runtime verification community suggests that they are indeed easy to write [12].

Our contributions are as follows. We formally define our specification language "monitor-triggered temporal logic". We show that the way we combine monitors with LTL indeed bypasses the need to reason about the state-space of monitors. Thus, avoiding some of the algorithmic challenges of synthesis. We briefly present our synthesis tool. We give examples highlighting the benefits of using monitors, focusing on counting (with appropriate counter variables updated by monitor transitions) and parametrisation (with unspecified parameter variables that can be instantiated to any required value). Full proofs of the propositions and theorems claimed can be found in the appendix.

Related Work In the literature we find several approaches that use monitors in the context of synthesis. Ulus and Belta use monitors with reactive control for robotic system navigation, with monitors used for lower-level control (e.g. to identify the next goal locations), and controllers used for high-level control to avoid conflicts between different robots [29]. Wenchao et al. consider human-in-the-loop systems, where occasionally the input of a human is required. The controller monitors the environment for any possible violations, and invokes the human operator when necessary [19].[1] The use of monitors in these approaches is *ad hoc*, a more general approach is that of the `Spectra` language [22]. Essentially, Spectra monitors have an initial state, and several safety transition rules of the

[1] We can think of our approach as dual, where the monitor invokes the synthesised controller when necessary.

form $p \rightarrow q$, where p is a proposition on some low-level variables, and q defines the `next` value of the monitor variable. This monitor variable can be used in the higher-level controller specification. The approach here is more general than ours in a sense, since we limit ourselves to using monitors as triggers, however our monitors are more succinct and expressive.

The notion of triggers in temporal logic is not new, with regular expressions being used as triggers for LTL formulas in different languages [2, 28, 10, 13]. Complexity wise, Kupferman et al. show how the synthesis of these trigger properties is 2EXPTIME-complete [18]. However, in order to support such logics algorithms would have to incorporate the entire state-space of the automata induced by the regular expression triggers. We are not aware of implementations supporting synthesis from such extensions of LTL. Using automata directly within the language, as we do, may be more succinct and convenient. We also include a *repetition* of trigger formulas in a way that is different from these extensions. However, the main difference is in avoiding the need to reason about the triggering parts.

Our combination of monitors and LTL formulas can be seen as a control-flow composition [20]. Lustig and Vardi discuss how to synthesise a control-flow composition that satisfies an LTL formula given an existing library of components. They consider all components to be given and synthesise the composition itself. Differently, we assume the composition to be given and synthesise a controller for the LTL part. Other work given a global specification reduces it according to that of the existing components, resulting in a specification for the required missing component [27]. This is at a higher level than our work, since we start with specifications for each component.

2 Preliminaries

We write σ for infinite traces over an event alphabet Σ. We use the notation $\sigma_{i,j}$, where $i, j \in \mathbb{N}$ and $i \leq j$, to refer to the sub-trace of σ starting from position i, ending at (including) position j. We write σ_i for $\sigma_{i,i}$, and $\sigma_{i,\infty}$ for the suffix of σ starting at i.

Linear Temporal Logic (LTL) General LTL (ϕ) and co-safety LTL (φ) are defined over a set of propositions $\mathcal{P}$ respectively as follows, where $e \in \mathcal{P}$:

$$\phi \stackrel{\text{def}}{=} \mathbf{tt} \mid \mathbf{ff} \mid e \mid \neg e \mid \phi \wedge \phi \mid \phi \vee \phi \mid X\phi \mid \phi U \phi \mid G\phi$$
$$\varphi \stackrel{\text{def}}{=} \mathbf{tt} \mid \mathbf{ff} \mid e \mid \neg e \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid X\varphi \mid \varphi U \varphi$$

We also define and use $F\phi \stackrel{\text{def}}{=} \mathbf{tt} U \phi$ and $\phi W \phi' \stackrel{\text{def}}{=} (\phi U \phi') \vee G\phi$. We write $\sigma \vdash \phi$ for $\sigma_{0,\infty} \vdash \phi$. We omit the standard semantics of LTL [24].[2]

Mealy Machines A *Mealy machine* is a tuple $C = \langle S, s_0, \Sigma_{in}, \Sigma_{out}, \rightarrow, F \rangle$, where S is the set of states, s_0 the initial state, Σ_{in} the set of input events, Σ_{out} the set of output events, $\rightarrow\colon S \times \Sigma_{in} \mapsto \Sigma_{out} \times S$ the complete deterministic

[2] This is also available in the Appendix A.

transition function, and $F \subseteq S$ a set of accepting states. For $(s, I, O, s') \in \rightarrow$ we write $s \xrightarrow{I/O} s'$.

Notice that by definition for every state $s \in S$ and every $I \in \Sigma_{in}$ there is $O \in \Sigma_{out}$ and s' such that $s \xrightarrow{I/O} s'$. A *run* of the Mealy machine C is $r = s_0, s_1, \ldots$ such that for every $i \geq 0$ we have $s_i \xrightarrow{I_i/O_i} s_{i+1}$ for some I_i and O_i. A run r *produces* the word $w = \sigma_0, \sigma_1, \ldots$, where $\sigma_i = I_i \cup O_i$. We say that C produces the word w if there exists a run r producing w. We say that C *accepts* a prefix u of w if $s_{|u|} \in F$.

Realisability An LTL formula φ over set of events $\mathcal{P} = \mathcal{P}_{in} \cup \mathcal{P}_{out}$ is *realisable* if there exists a Mealy machine C over input events $2^{\mathcal{P}_{in}}$ and output events $2^{\mathcal{P}_{out}}$ such that for all words w produced by C we have $w \vdash \varphi$. We say C realises φ.

Theorem 1 ([26]). *Given an LTL formula φ it is decidable in 2EXPTIME whether φ is realisable. If φ is realisable the same algorithm can be used to construct a Mealy machine C_φ realising φ.*

2.1 Flagging Monitors

We introduce our own simplified version of DATEs [8, 3], *flagging monitor*s, as a formalism for defining runtime monitors. Flagging monitors (monitors, for short) are different from DATEs in that they work in discrete time, and events are in the form of sets. Monitors are designed such that once they *flag* (accept) they never flag again. This is modeled by having *flagging* states, which are used to signal that monitoring has ended successfully. We also use *sink* states, from which it is assured the monitor cannot flag in the future. We ensure that the monitor flags only upon determining a matching sub-trace, and thus a monitor upon reaching a flagging state can never flag again.

Monitor A *monitor* is a tuple $D = \langle \Sigma, \mathbb{V}, \Theta, Q, \theta_0, q_0, F, \bot, \rightarrow \rangle$, where Σ is the event alphabet, $\mathbb{V}$ is a set of typed variables, Θ is the set of possible valuations of $\mathbb{V}$, Q is a finite set of states, $\theta_0 \in \Theta$ is the initial variable valuation, $q_0 \in Q$ is the initial state, $F \subseteq (Q \setminus \{q_0\})$ is the set of *flagging states* (we often use $q_F \in F$), $\bot \in Q$ is a *sink* state, and $\rightarrow \in Q_\top \times (\Sigma \times \Theta \mapsto \{true, false\}) \times (\Sigma \times \Theta \mapsto \Theta) \mapsto Q$ is the *deterministic transition function*, from $Q_\top \stackrel{\text{def}}{=} Q \setminus \{\bot\}$, activated if a guard holds on the input event and the current variable valuation, while it may perform some action to transform the valuation.

For $(q, g, a, q') \in \rightarrow$ we write $q \xrightarrow{g \mapsto a} q'$, and we will be using E as the input event parameter for both g and a. We omit g when it is the *true* guard, and a when it is the *null* action. We use $D_{\langle * \rangle}$ for the monitor that accepts on every event, i.e. $\langle \Sigma, V, \Theta, \{q_0, q_F, \bot\}, \theta_0, q_0, \{q_F\}, \bot, \{q_0 \xrightarrow{true \mapsto null} q_F\} \rangle$.

For example, the monitor in Figure 1 keeps a counter that counts the number of *knock* events, and flags when the number of knocks is exactly n.

We give an operational semantics to *monitor*s, with configurations as pairs of states and valuations, with transitions between configurations tagged by events.

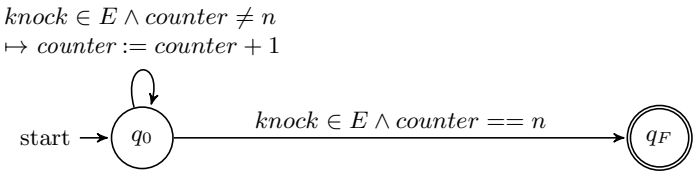

Fig. 1. Monitor that counts the number of knocks, and flags after n knocks.

Monitor Semantics The semantics of flagging monitors [3] is given over configurations of type $Q \times \Theta$, with transitions labeled by Σ, and the transition $\rightarrow$ defined by the following rules: (1) A transition from a non-flagging and non-sink configuration is taken when the guard holds on the event and valuation, and then the latter is updated according to the transition's action; (2) If there is no available transition whose guard holds true on the current valuation then transition to the same configuration (stutter); (3) A sink configuration cannot be left; and (4) A flag configuration always transitions to the sink configuration. We use $\Rightarrow$ for the transitive closure of $\rightarrow$.[3]

Flagging Trace A finite trace is said to be flagging if it reaches a flagging state. $\sigma_{i,j} \Vdash D \stackrel{\text{def}}{=} \exists q_F, \theta' \cdot (q_0, \theta_0) \stackrel{\sigma_{i,j}}{\Rightarrow} (q_F, \theta')$.

The semantics ensures that every extension of a flagging trace is non-flagging.

Proposition 1. $\forall \sigma \in \Sigma^\omega \cdot \forall n \in \mathbb{N} \cdot \sigma_{i,j} \Vdash D \wedge n > 0 \implies \sigma_{i,j+n} \not\Vdash D$.

We can also easily show that $D_{\langle * \rangle}$ accepts all traces of length one.

Proposition 2. $\forall \sigma \in \Sigma^\omega$ *and* $\forall i \in \mathbb{N} \cdot \sigma_{i,i} \Vdash D_{\langle * \rangle}$.

3 Monitors as Triggers for LTL Formulas

We suggest a simple kind of interaction between monitors and LTL, where monitors are used as *triggers* for LTL. Previous work has considered the use of a trigger operator that activates the checking of an LTL expression when a certain regular expression matches [2]. Our approach here is similar, except that we maintain a stricter separation between the monitored and temporal logic parts.

Our language combining monitors with LTL has three operators: (i) monitors as a trigger for an LTL formula; (ii) repetition of the trigger formula (when the LTL formula is co-safety); and (iii) assumptions in the form of LTL formulas.

Definition 1 (Monitor-Triggered Temporal Logic). *Monitor-triggered temporal logic extends LTL with three operators:*

$$\begin{aligned} \pi' &= D{:}\phi \mid (D; \varphi)^* \\ \pi &= \phi \rightarrow \pi' \end{aligned}$$

[3] See Definition 7 in Appendix A for full formal semantics.

Formula $D{:}\phi$ denotes the triggering of an LTL formula ϕ by a monitor D. We call formulas of this form simple-trigger LTL. *Formula $(D;\varphi)^*$ repeats infinitely the triggering of a co-safety LTL formula φ by a monitor D. We call formulas of this form* repeating-trigger LTL. *Finally, $\phi \rightarrow \pi'$ models a specification with an LTL assumption ϕ. LTL formulas are defined over a set of propositions $\mathcal{P} = \mathcal{P}_{in} \cup \mathcal{P}_{out}$ and monitors over the alphabet $\Sigma = 2^{\mathcal{P}_{in}}$.*

In formulas of the form $D{:}\phi$ if D flags then the suffix must satisfy ϕ. In formulas of the form $(D;\varphi)^*$, the monitor restarts after satisfaction of the co-safety formula φ. For example, if D is the monitor in Figure 1 and $\varphi = (\mathit{open} \wedge X(\mathit{greet} \wedge X\,\mathit{close}))$, then $D{:}\varphi$ would accept every trace that waits for knocks, and at the nth knock opens the door, then greets, and then closes the door. On the other hand, $(D;\varphi)^*$ requires the trace to arbitrarily repeat this behaviour.

To support repeating triggers, we define the notion of tight satisfaction.

Definition 2 (Tight Co-Safety LTL Satisfaction). *A finite trace is said to* tightly *satisfy a co-safety LTL formula if it satisfies the formula and no strict prefix satisfies the formula: $\sigma_{i,j} \Vdash \varphi \stackrel{\text{def}}{=} \sigma_{i,j} \vdash \varphi \wedge (\forall k \cdot i \leq k < j \implies \sigma_{i,k} \nvdash \varphi)$. We also call such a trace a tight witness for the LTL formula.*

Note that here a tight witness is not necessarily a minimal witness (in the sense that all of its extensions satisfy the LTL formula). For example, for every set of propositions P, a trace $\langle P \rangle$ is a minimal witness for $X\mathbf{tt}$ [5]. However it is not a tight witness in our sense, since $\langle P \rangle \nvdash X\mathbf{tt}$. On the other hand $\langle P, P \rangle$ is a tight witness since $\langle P, P \rangle \vdash X\mathbf{tt}$ and every prefix of it does not satisfy $X\mathbf{tt}$.

Notice that it would not be simple to just use finite trace semantics for full LTL [11, 15, 10, 4, 5]. Consider for example, the trace $\langle \{a\} \rangle$, which satisfies Ga. It is not clear how to define tight satisfaction in order to start the monitor again. For example, $\langle \{a\} \rangle$ can be extended to $\langle \{a\}, \{a\} \rangle$ and still satisfy Ga. Hence formulas of the form $(D;\varphi)^*$ are restricted to co-safety LTL, where satisfaction over finite traces is well-defined and accepted.

We now define the trace semantics of the trigger and repetition operators.

Definition 3 (Monitor-Trigger Temporal Logic Semantics).

1. *An infinite trace satisfies a simple-trigger LTL formula if when a prefix of it causes the monitor to flag then the corresponding suffix (including the last element of the prefix) satisfies the LTL formula:*

$$\sigma_{i,\infty} \vdash D{:}\phi \stackrel{\text{def}}{=} \exists j \cdot i \leq j \wedge (\sigma_{i,j} \Vdash D \implies \sigma_{j,\infty} \vdash \phi) \qquad \text{where } i \in \mathbb{N}.$$

2. *A finite trace satisfies one step of a repeating-trigger LTL formula if a prefix of it causes the monitor to flag and the corresponding suffix (including the last element of the prefix) tightly satisfies the co-safety LTL formula:*

$$\sigma_{i,k} \vdash D;\varphi \stackrel{\text{def}}{=} \exists j \cdot i \leq j \leq k \wedge (\sigma_{i,j} \Vdash D \wedge \sigma_{j,k} \Vdash \varphi) \qquad \text{where } i, k \in \mathbb{N}.$$

3. *An infinite trace satisfies a repeating-trigger LTL formula if when a prefix of it matches the monitor then the corresponding infinite suffix matches the LTL formula:*

$$\sigma \vdash (D;\varphi)^* \stackrel{\text{def}}{=} \forall i \cdot \sigma_{0,i} \Vdash D \implies \exists j \cdot j \geq i \wedge \sigma_{0,j} \vdash D;\varphi \wedge \sigma_{j+1,\infty} \vdash (D;\varphi)^*.$$

4. *An infinite trace satisfies a specification π' with an assumption ϕ when if it satisfies ϕ it also satisfies π':*

$$\sigma \vdash \phi \rightarrow \pi' \stackrel{\text{def}}{=} \sigma \vdash \phi \implies \sigma \vdash \pi'.$$

An interesting aspect of this semantics is that in a formula $D;\varphi$, D and φ share an event, and the same for $D{:}\phi$. This is a choice we make to allow for message-passing between the two later on. Here it does not limit us, since not sharing a time step can be simulated by adding a further transition with a *true* guard before flagging, or by simply transforming ϕ into $X\phi$.

This semantics ensures that given an infinite trace, when a finite sub-trace satisfies $D;\varphi$, extensions of the sub-trace do not also satisfy it.

Proposition 3. $\sigma_{i,j} \vdash D;\varphi \implies \forall k > j \cdot \sigma_{i,k} \nvdash D;\varphi$.[4]

We can prove that a trace σ satisfies an LTL formula ϕ iff it also satisfies the formula where ϕ is triggered by the empty *monitor*.

Proposition 4. $\sigma \vdash \phi \iff \sigma \vdash D_{\langle * \rangle}{:}\phi$.[5]

Moreover, we can show that adding these monitors as triggers for LTL formulas results in a language that is more powerful than LTL.

Theorem 2. *Our language is strictly more expressive than LTL.*

Proof. Proposition 4 shows that every LTL formula ϕ can be written in our language as $D_{\langle * \rangle};\phi$. LTL cannot express the property that each even time step must have p be true [30] (regardless of what is true at odd steps). In our language $(D_{\langle * \rangle};p \wedge X\mathbf{tt})^*$ specifies that p is true in every even time step, and $(D_{\langle * \rangle};Xp)^*$ specifies that p is true in every odd time step. □

Our logic is even more expressive, for example Figure 2 shows a monitor that flags upon the average occurrence of an event falling below a certain level. We note that, in general, we have not restricted the types of variables of a monitor to range over finite domains. Thus, a monitor could also identify context-free or context-sensitive languages or, indeed, be Turing powerful. However, Theorem 2 holds even if we consider only monitors whose variables have finite domains, or even monitors without variables.

[4] Proof can be found in Appendix B.
[5] Proof can be found in Appendix B.

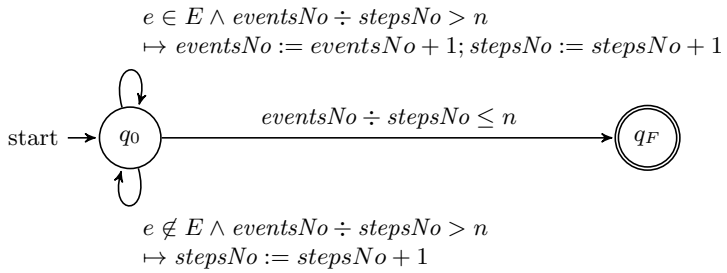

Fig. 2. Monitor that keeps track of the number of time steps, and the number of occurrences of e, while flagging is the average occurrence of e goes below n.

4 Synthesising Monitor-Triggered Controllers

We have so far discussed our language from a satisfaction viewpoint. However we are interested in synthesising systems that enforce the specifications in our language. In this section we present our synthesis approach, which relies on the synthesis of controllers for LTL formulas.

Consider a specification $\pi = \gamma \rightarrow \pi'$, where π' is either of the form $D{:}\phi$ or $(D;\varphi)^*$. We focus on specifications where the assumption γ is restricted to conjunctions of simple invariants, transition invariants, and recurrence properties. Formally, we have the following:

$$\alpha \stackrel{\text{def}}{=} \mathbf{tt} \mid \mathbf{ff} \mid a \mid \neg\alpha \mid \alpha \wedge \alpha \mid \alpha \vee \alpha$$
$$\beta \stackrel{\text{def}}{=} \alpha \mid X\alpha \mid \beta \wedge \beta \mid \beta \vee \beta$$
$$\gamma \stackrel{\text{def}}{=} G\beta \mid GF\alpha \mid \gamma \wedge \gamma$$

That is, α are Boolean combinations of propositions, β allows next operators without nesting them, and γ is a conjunction of invariants of Boolean formulas, Boolean formulas that include next, or recurrence of Boolean formulas. We discuss below the case of general assumptions.

Let $\pi = \gamma \rightarrow (D{:}\phi)$. Then $t(\pi)$ is the formula $\gamma \rightarrow \phi$. Let $\pi = \gamma \rightarrow (D;\varphi)^*$. Then $t(\pi)$ is the formula $\gamma \rightarrow \varphi$. That is, $t(\pi)$ is the specification obtained by considering the implication of the assumption γ and the LTL formula.

4.1 Tight Synthesis for co-Safety Implication Formulas

Let π contain a repeating trigger and let $t(\pi) = \gamma \rightarrow \varphi$, where φ is a co-safety formula. Suppose that $t(\pi)$ is realisable and let $C_{t(\pi)}$ be a Mealy machine realising $t(\pi)$.

Definition 4. *A Mealy machine C* tightly realises *a formula of the form $\gamma \rightarrow \varphi$, where φ is a co-safety formula, if it realises $\gamma \rightarrow \varphi$ and in addition for every word w produced by C such that $w \vdash \gamma$ there exists a prefix u of w such that C accepts u, $u_{0,|u|} \Vdash \varphi$, and for every $u' < u$ we have C does not accept u.*

That is, when the antecedent γ holds, the Mealy machine accepts the *tight* witness for satisfaction of φ.

Theorem 3. *The formula $t(\pi) = \gamma \rightarrow \varphi$ is tightly realisable iff it is realisable. A Mealy machine tightly realising $t(\pi)$ can be constructed from $C_{t(\pi)}$ with the same complexity.*

Proof (sketch). We construct a deterministic finite automaton that is at most doubly exponential in ϕ, that accepts all finite prefixes that satisfy ϕ. Its product with $C_{t(\pi)}$ results in a Mealy machine that accepts all prefixes that satisfy $t(\pi)$, in particular the shortest prefix, as required for realisability.[6]

Note that in the case of tight realisability we can give a controller with a set of accepting states that enable us to accept upon observing tight witnesses. In the case where we are only concerned about non-tight realisability we assume the controller does not have any accepting states.

4.2 Monitor-Triggered Synthesis

We are now ready to handle synthesis for monitor-triggered LTL.

Definition 5. *A monitor-triggered LTL formula π over set of events $\mathcal{P}_{in}$ and $\mathcal{P}_{out}$ is* realisable *if there exists a Mealy machine C over input events $2^{\mathcal{P}_{in}}$ and output events $2^{\mathcal{P}_{out}}$ such that for all words w produced by C we have $w \vdash \pi$. We say that C* realises *π.*

In the case of simple triggers, we combine the monitor with a Mealy machine realising $t(\pi)$. In the case of repeating triggers, we combine the monitor with a Mealy machine tightly realising $t(\pi)$. In what follows we define the behaviour of the combination of a monitor and a Mealy machine.

Consider a specification $\pi = \gamma \rightarrow \pi'$, where π' is either $M{:}\phi$ or $(M; \varphi)^*$.

Theorem 4. *Let $C_{t(\pi)}$ be a Mealy machine realising $t(\pi)$ when π' is a simple-trigger LTL, and tightly realising $t(\pi)$ when π' is a repeating-trigger LTL. Then there is a Mealy machine $M \blacktriangleright C_{t(\pi)}$ that realises π.*

Proof (sketch). $M \blacktriangleright C_{t(\pi)}$ can be constructed over states that correspond to a tuple of M states, valuations, and $C_{t(\pi)}$ states. Monitor transitions can be unfolded into Mealy machine transitions with no outputs, according to their semantics. Transitions to a flagging state can be composed with transitions from the initial state of $C_{t(\pi)}$. For the repeating case, transitions to final states of $C_{t(\pi)}$ are made instead to point back to the initial configuration (initial state and valuation of M, and initial state of $C_{t(\pi)}$). Execution happens only in one machine at a time, except for the shared transition in the repeating case. We show by induction the correctness of this construction.[7]

[6] Full proof can be found in Appendix C.
[7] Full proof can be found in Appendix C.

The opposite of Theorem 4 is, however, not true. If π is realisable then it does not necessarily mean that $t(\pi)$ is also realisable. Consider a specification with a monitor that never flags, and which thus any Mealy machine realises. Another example is with a monitor that only flags upon seeing the event set $\{a\}$, and an LTL formula of the form $(b \implies \mathbf{ff}) \wedge (a \implies c)$ (where a and b are input events, and c an output event). The LTL formula is clearly unrealisable given the first conjunct, however the combination of the monitor with a controller for LTL's second conjunct would realise the corresponding specification. Thus the construction in Thereom 4 is only sound but not complete, i.e., we have a procedure to produce controllers for our language only when the underlying LTL formula (modulo the assumption) is realisable, or when the monitor cannot flag.

Corollary 1. *If M cannot flag, or $t(\pi)$ is realisable, then π is realisable.*

We recall that we have restricted the assumptions to a combination of invariants, transition invariants, and recurrence properties. Such assumptions are "state-less". That is, identifying whether a word satisfies an assumption does not require to follow the state of the assumption. Thus, in our synthesis procedure it is enough for the controller to check whether the assumption holds without worrying about what happened during the run of the monitor that triggered it. In particular, if (safety) assumptions are violated only during the run of monitors, our Mealy machine will still enforce satisfaction of the implied formula. In order to treat more general assumptions, we would have to either analyse the structure of the monitor in order to identify in which "assumption states" the controller could be started or give a precondition for synthesis by requiring that the controller could start from an arbitrary "assumption state", which we leave for future work. Similarly, understanding the conditions the monitor enforces and using them as assumptions would allow us to get closer to completeness of Theorem 4. One coarse abstraction is simply the disjunction of the monitor's flagging transitions' guards as initial assumptions for the LTL formula.

5 Tool Support

We created a proof-of-concept automated tool[8] to support the theory presented in this paper. Implemented in Python, this tool currently accepts as input a monitor written in a syntax inspired by that of LARVA [9, 3], and an LTL specification, while it outputs a symbolic representation of the Mealy machine constructed in the proof of Theorem 4, in the form of a monitor with outputs.

The proof of Theorem 3 is constructive and provides an optimal algorithm to synthesise tight controllers using standard automata techniques. For this tool we have instead opted to re-use an existing synthesis tool, Strix [21, 23] due to its efficiency. To force Strix to synthesise a tight controller (for repeating triggers), the tool performs a transformation to the co-safety guarantees to output a new event that is only output once a tight witness is detected. This transformation

[8] https://github.com/dSynMa/syMTri

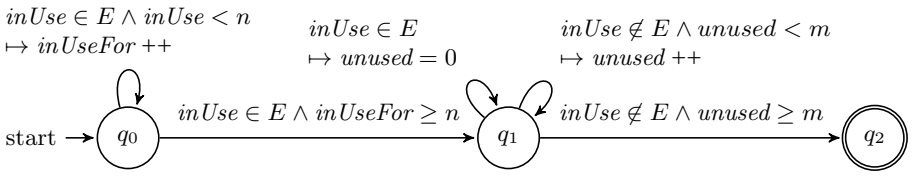

Fig. 3. Flagging monitor that checks that the room has been in use for n time steps, after which when there is a period of m time steps where the room is empty it flags.

works well on our case studies, but is exponential in the worst-case. This is due to the need for disambiguating disjunctions. For example, given $\psi_1 \vee \psi_2$ we cannot in general easily be sure which disjunct the controller will decide to enforce; instead we disambiguate it to $(\psi_1 \wedge \neg\psi_2) \vee (\neg\psi_1 \wedge \psi_2) \vee (\psi_1 \wedge \psi_2)$ (cf. [7]).

6 Case Studies

We have applied our monitor approach mainly in the setting of conditions on the sequence of environment events, for which synthesis techniques can be particularly inefficient. We will consider a case study involving such conditions, where several events need to be observed before a robot can start cleaning a room. Furthermore we consider a problem from SYNTCOMP 2020 on which all tools timed out due to exponential blowup as the parameter values increase, relating to observing two event buses. We show how our approach using monitors avoids the pitfalls of existing approaches with regards to these kinds of specifications.

6.1 Event Counting

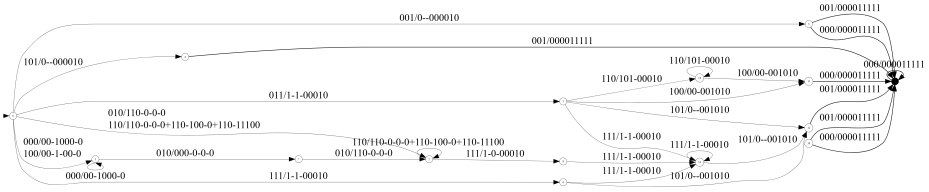

Fig. 4. Tight controller for cleaning robot, with rightmost state as accepting state. (1 (0) in position i means event i (not) occurs, and - when we do not care).

Consider a break room that is used by people intermittently during the day, and that needs to be cleaned periodically by a cleaning robot. We do not want to activate the robot every time the room is unclean to not disturb people on their break. Instead our procedure involves checking that the room is in use for

a certain amount n of time steps. We also do not want the cleaning robot to be too eager or to activate immediately upon an empty room. Thus we further want to constrain the robot's activation on the room being empty for a number of m time steps and reset the counting whenever the room is not empty. We can represent these conditions using the monitor in Figure 3.

Given a set of assumptions on the environment (e.g. cleaning an unclean locked room eventually results in a clean room), we wish the controller to satisfy that eventually the room will be clean, after which the robot leaves the room and opens the door to the public: $F(\mathit{isClean}\ \&\ (XF\ !\mathit{inRoom})\ \&\ (XF\ !\mathit{doorLocked}))$.[9] Our tool synthesises Figure 4 as a tight controller for this.

Representing the first monitor condition in LTL is not difficult ($\neg pW(p \wedge X(\neg pWp \wedge ...))$), where proposition p corresponds to *inRoom* and W is the weak until operator. The second condition is different, given the possible resetting of the count, but still easily representable in LTL ($(\bigvee_{i=0}^{m-1} X^i p)W(\bigwedge_{i=0}^{m-1} X^i \neg p)$). Setting $n, m = 2$, and φ to be what we require out of the cleaning robot in one step, then a step of our specification (*without repetition*) in LTL is:

$$\psi = \neg pW(p \wedge X(\neg pW(p \wedge X((p \vee Xp)W(\neg p \wedge X(\neg p \wedge \varphi)))))).$$

In fact Strix confirms this to be realisable, and produces an appropriate Mealy Machine with eighty transitions, the size of which increases with each increase in any of the parameters.

However, using our approach all we require is Figure 3 and Figure 4. By representing the counting part of the specification using a monitor we can create a specification much more succinct than the LTL one, while its representation is of the same size for each value of the parameter. Moreover in LTL it is not clear how to reproduce our repeating triggers.

The difference is that the traditional approaches explicitly enumerate every possible behaviour and state of the controller at runtime, which can get very large. In our approach we are instead doing this symbolically, and allowing the particular behaviour of the environment at runtime to drive our symbolic monitor. The extra cost associated with this is the semantics of guard evaluation and maintaining variable states. For this example, the cost of the variable states (only two variables) is much smaller than the cost of the Strix generated machine, while guards simply check for membership and use basic arithmetic operations.

6.2 Sequences of Events

We consider a benchmark from SYNTCOMP 2020 [1][10], that generates formulas of the form, e.g. for $n = 2$, $F(p_0 \wedge F(p_1)) \wedge F(q_0 \wedge F(q_1)) \iff GFacc$. Strix [21], the best-performing tool in the LTL tracks of the competition, was successful when the bus size was small, however timed out for $n = m = 12$ (and above). The issue here is that the generated strategy must take into account every possible interleaving of the two sequences, which quickly causes a state space explosion.

[9] The full specification is available with our tool.

[10] The considered benchmark corresponds to files of the form `ltl2dba_beta_<n>.tlsf`.

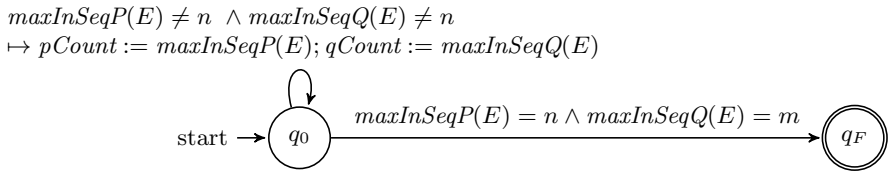

Fig. 5. Event ordering in two buses.

With our approach we can represent the left-hand side in constant-size for any n and m as illustrated in Figure 5, where *maxInSeqX* is a function that when $xCount = i$ returns i if E does not contain x_{i+1}, and otherwise returns the maximal j such $x_{i+1}, x_{i+2}, ...$, and x_j are all in E. The benefits apply however complex the right-hand side.

To replicate the whole LTL formula we can use M:$GFacc$, where M is the monitor in Figure 5. This is somewhat different from the original specification, where a necessary and sufficient relation was specified. One would be tempted to specify this as (for $n = 1$) $(M; (Facc))^*$, however the monitor is not active while the controller is activated, thus p_1 and q_1 may occur in tandem with *acc* but be missed by the monitor. Although this is not of consequence towards the satisfaction of the formula here (p_1 should occur infinitely often), this is not generally the case. On the other hand M:$GFacc$ captures that upon the first activation of M there is no need to monitor the environment's behaviour anymore, and thus is equivalent to the original specification for control.

7 Discussion

The case studies we considered in the previous section focused on counting and waiting for sequence of events. We expect other useful applications of monitors as triggers, given they can be used to specify more sophisticated quantitative properties out of reach for LTL, e.g. see Figure 2 [9].

We have highlighted how our approach extends the scope of use of reactive synthesis. It is clear that we can gain in scalability and expressiveness, but there is a price to pay: the "trigger" part. In general, to avoid lack of guarantees one can avoid working directly with automata, and instead use regular expressions or co-safety LTL formulas (under our notion of tight satisfaction) as triggers. Standard inexpensive monitor synthesis [6] could then be used to generate a monitor. In the case of more expressive manually-written monitors, which is standard for runtime verification (e.g. [8]), in practice one can easily apply model checking to the monitor to ensure it satisfies specific properties (e.g. no infinite loops).

There are certain benefits to using a symbolic representation, including succinct representation, and easy parametrisation. The Mealy machine construction we give in the proof of Theorem 4 is in fact not carried out by our tool, but instead it produces a symbolic monitor with outputs that essentially performs the construction on-the-fly. The cost of unfolding is then only paid for the trace at

runtime, rather than for all possible traces. Moreover, a symbolic representation allows for specification of parametrised specifications, when parametrisation can be pushed to the monitor side. This can be done by adding any required parameter to the variables of the monitor, and instantiating its value in the initial variable valuation of the monitor appropriately. Note that our results are agnostic of the initial valuation, and thus hold regardless of the parameter values.

We have not yet discussed conjunction of trigger formulas, e.g. $(M_1; \psi_1)^* \wedge (M_2; \psi_2)^*$. Conjunction is easy when the output events of ψ_1 and ψ_2 respectively talk about are independent from each other. Our controller construction can be used independently for each. Similarly, when the properties are safety properties there is no difficulty. However, when, e.g., ψ_1 is a liveness property with at least one output event correlated with an output event of ψ_2, then conjunction is more difficult, due to possible interaction between the two possibly concurrent controllers. We are investigating a solution for this issue of concurrency of controllers by identifying appropriate assumptions about the monitor.

Theorem 2 compares our expressive power to that of LTL. We also mention that we do not restrict the variables used by monitors. Thus, even when comparing with languages that include regular expressions or automata [2, 10, 18] our language would be more expressive. If we were to restrict monitors to be finite state, then, as these languages can express all ω-regular languages, it is clear that they would be able to express our specifications. We note, however, that the repeating trigger operator is not directly expressible in these languages. Thus, the translation involves a conversion of our specification to an automaton and embedding this automaton in "their" specification. The conversion of our specification to an automaton includes both the enumeration of the states of the monitor and the exponential translation of LTL to (tight) automata.

8 Conclusions

We have explored synthesis for specifications that combine modelling and declarative aspects, in the form of symbolic monitors triggers for LTL formulas. We have shown how this extends the scope of synthesis by allowing parts of a specification that are hard for synthesis to be instead handled in the monitor part. The synthesis algorithm we give synthesises the LTL part without requiring the need to reason about the monitor. Moreover, we have implemented this approach and applied it to several case studies involving counting and monitoring multiple sequences of events that can be impossible or hard for LTL synthesis. We showed how by exploiting the symbolic nature of the monitors we can create fixed-size parameterised controllers for some parameterised specifications.

Future Work Our work opens the door to a number of interesting research avenues, both by using richer monitor triggers and by exploring different interactions between triggers and controllers. We discuss below just a few such possibilities. In all the cases below the challenges lie not only in providing a new

language to capture the extension but rather in the theoretical framework with a proof that the integration is sound.

A first intuitive extension is to add real-time to the monitors, to express properties like "*compute the average use of a certain resource every week and activate the controller to act differently depending on whether the average is bigger (or smaller) than a certain amount*". While extending the monitor with real-time is quite straightforward (our monitors are restricted versions of DATEs [8] which already contain timers and stopwatches), the challenge will be to combine it with the controller in a suitable manner. Having real-time monitors running in parallel with controllers would enable for instance the possibility to add timeouts to activities performed by the controllers.

Currently we have a strict alternation between the execution of the monitor and the controller: we would like to explore under which conditions the two can instead run in parallel. This would allow the controller to react to the monitor only when certain complex condition hold while the controller is active doing other things (e.g., the monitor might send an interruption request to the controller when a certain sequence of events happens within a certain amount of time, while the controller is busy ensuring a fairness property).

We could also have many triggers that run in parallel activating different controllers, or even some meta-monitor that acts as an orchestrator to enable and disable controllers depending on certain conditions. This might require to extend/modify the semantics since the interaction might be done asynchronously.

We would like to address the limitation of controller synthesis concerning what to do when the assumptions are not satisfied. It is well-known that in order to be able to automatically synthesise a controller very often one must have strong assumptions, and nothing is said in case the assumptions are not satisfied. We would like to explore the use of monitors to monitor the violation of assumptions and interact with the controller in order to coordinate how to handle those situations (we can for instance envisage a procedure that automatically extends the controller with transitions that takes the controller to a recovery state if the assumptions are violated).

References

1. Syntcomp 2020, http://www.syntcomp.org/syntcomp-2020-results/
2. Armoni, R., Fix, L., Flaisher, A., Gerth, R., Ginsburg, B., Kanza, T., Landver, A., Mador-Haim, S., Singerman, E., Tiemeyer, A., Vardi, M., Zbar, Y.: The forspec temporal logic: A new temporal property-specification language. In TACAS. LNCS, Springer 2002.
3. Azzopardi, S., Ellul, J., Pace, G.J.: Monitoring smart contracts: Contractlarva and open challenges beyond. In RV. LNCS, Springer 2018.
4. Bartocci, E., Bloem, R., Nickovic, D., Roeck, F.: A counting semantics for monitoring LTL specifications over finite traces. In CAV. LNCS. Springer 2018.
5. Bauer, A., Leucker, M., Schallhart, C.: The good, the bad, and the ugly, but how ugly is ugly? In RV. LNCS, Springer 2007.
6. Bauer, A., Leucker, M., Schallhart, C.: Runtime verification for ltl and tltl. ACM Trans. Softw. Eng. Methodol. **20**(4) (2011).

7. Benedikt, M., Lenhardt, R., Worrell, J.: LTL model checking of interval markov chains. In TACAS. LNCS. Springer 2013.
8. Colombo, C., Pace, G.J., Schneider, G.: Dynamic event-based runtime monitoring of real-time and contextual properties. In FMICS. LNCS. Springer 2009.
9. Colombo, C., Pace, G.J., Schneider, G.: LARVA — Safer Monitoring of Real-Time Java Programs. In SEFM. IEEE 2009.
10. De Giacomo, G., Vardi, M.Y.: Linear temporal logic and linear dynamic logic on finite traces. In IJCAI. AAAI Press 2013.
11. Eisner, C., Fisman, D., Havlicek, J., Lustig, Y., McIsaac, A., Van Campenhout, D.: Reasoning with temporal logic on truncated paths. In CAV. LNCS, Springer 2003.
12. Falcone, Y., Krstić, S., Reger, G., Traytel, D.: A taxonomy for classifying runtime verification tools. In RV. LNCS. Springer 2018.
13. Faymonville, P., Zimmermann, M.: Parametric linear dynamic logic. Inf. Comput. **253**, 237–256 (2017).
14. Filippidis, I., Murray, R.M., Holzmann, G.J.: A multi-paradigm language for reactive synthesis. In SYNT. EPTCS 2015.
15. Fisman, D., Kugler, H.: Temporal reasoning on incomplete paths. In ISoLA. LNCS. Springer 2018.
16. Kupferman, O.: Automata theory and model checking. In: Handbook of Model Checking. Springer 2018.
17. Kupferman, O., Vardi, M.Y.: Model checking of safety properties. Formal Methods Syst. Des. **19**(3), 291–314 (2001)
18. Kupferman, O., Vardi, M.Y.: Synthesis of trigger properties. In LPAR. LNCS. Springer 2010.
19. Li, W., Sadigh, D., Sastry, S.S., Seshia, S.A.: Synthesis for human-in-the-loop control systems. In TACAS. LNCS. Springer 2014.
20. Lustig, Y., Vardi, M.Y.: Synthesis from component libraries. In FoSSaCS. LNCS. Springer 2009.
21. Luttenberger, M., Meyer, P.J., Sickert, S.: Practical synthesis of reactive systems from LTL specifications via parity games. Acta Informatica **57**(1-2), 3–36 (2020)
22. Maoz, S., Ringert, J.O.: Spectra: A specification language for reactive systems (2019)
23. Meyer, P.J., Sickert, S., Luttenberger, M.: Strix: Explicit reactive synthesis strikes back! In CAV. LNCS. Springer 2018.
24. Piterman, N., Pnueli, A.: Temporal logic and fair discrete systems. In: Handbook of Model Checking. Springer (2018)
25. Piterman, N., Pnueli, A., Sa'ar, Y.: Synthesis of reactive(1) designs. In: VMCAI. LNCS. Springer 2006.
26. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: POPL. ACM Press 1989.
27. Raclet, J.: Residual for component specifications. Electron. Notes Theor. Comput. Sci. **215**, 93–110 (2008).
28. Sistla, A.P., Wolfson, O.: Temporal triggers in active databases. IEEE Transactions on Knowledge and Data Engineering **7**(3), 471–486 (1995)
29. Ulus, D., Belta, C.: Reactive control meets runtime verification: A case study of navigation. In RV. LNCS. Springer 2019.
30. Wolper, P.: Temporal logic can be more expressive. Inf. Control. **56**(1/2), 72–99 (1983)

A Full Definitions and Proofs for Section 2 (Background)

Definition 6 (General LTL Satisfaction).

$$
\begin{aligned}
\sigma_{i,j} \vdash \boldsymbol{tt} &\stackrel{\text{def}}{=} true \\
\sigma_{i,j} \vdash \boldsymbol{ff} &\stackrel{\text{def}}{=} false \\
\sigma_{i,j} \vdash e &\stackrel{\text{def}}{=} e \in \sigma_{i,i} \\
\sigma_{i,j} \vdash \neg\phi &\stackrel{\text{def}}{=} \neg(\sigma_{i,j} \vdash \phi) \\
\sigma_{i,j} \vdash \phi \wedge \phi' &\stackrel{\text{def}}{=} (\sigma_{i,j} \vdash \phi) \wedge (\sigma_{i,j} \vdash \phi') \\
\sigma_{i,j} \vdash \phi \vee \phi' &\stackrel{\text{def}}{=} (\sigma_{i,j} \vdash \phi) \vee (\sigma_{i,j} \vdash \phi') \\
\sigma_{i,j} \vdash X\phi &\stackrel{\text{def}}{=} j > i \wedge \sigma_{i+1,j} \vdash \phi \\
\sigma_{i,j} \vdash \phi U \phi' &\stackrel{\text{def}}{=} \exists l \cdot l \leq j \wedge \sigma_{l,\infty} \vdash \phi' \wedge \forall k \cdot i \leq k < l \wedge \sigma_{k,j} \vdash \phi \\
\sigma_{i,j} \vdash G\phi &\stackrel{\text{def}}{=} j = \infty \wedge \forall k \cdot k \geq i \implies \sigma_{k,j} \vdash \phi
\end{aligned}
$$

Definition 7. *The semantics of flagging monitors [3] is given over configurations of type $Q \times \Theta$, with transitions labeled by Σ, and the transition $\rightarrow$ defined by the following rules:*

1. *A transition from a non-flagging and non-sink configuration is taken when the guard holds on the event and valuation, and the latter updated according to the transition's action:*

$$\frac{q_1 \xrightarrow{g \mapsto a} q_2 \qquad g(E,\theta) \qquad q_1 \notin F \cup \{\bot\}}{(q_1,\theta) \xrightarrow{E} (q_2, a(E,\theta))}$$

2. *If there is no available transition whose guard holds true on the current valuation then transition to the same configuration (stutter):*

$$\frac{\nexists q_1 \xrightarrow{g \mapsto a} q_2 \cdot g(E,\theta)}{(q_1,\theta) \xrightarrow{E} (q_1,\theta)}$$

3. *A sink configuration cannot be left:*

$$\frac{}{(\bot,\theta) \xrightarrow{E} (\bot,\theta)}$$

4. *A flag configuration always transitions to a sink configuration.*

$$\frac{q_F \in F}{(q_F,\theta) \xrightarrow{E} (\bot,\theta)}$$

Proposition 1. $\forall \sigma \in \Sigma^\omega \cdot \forall n \in \mathbb{N} \cdot \sigma_{i,j} \Vdash D \wedge n > 0 \implies \sigma_{i,j+n} \not\Vdash D$.

Proof. If $\sigma_{i,j} \Vdash D$, then $(q_0,\theta_0) \stackrel{\sigma_{i,j}}{\Rightarrow} (q_F,\theta')$. By the last rule of the semantics Def. 2.1 then $(q_0,\theta_0) \stackrel{\sigma_{i,j+n}}{\Rightarrow} (\bot,\theta')$, and by the third rule of the semantics $(\bot,\theta')$ is a sink state, and thus q_F cannot be reached. Thus no extension can satisfy D, since satisfaction requires exactly reaching q_F.

Proposition 2. $\forall \sigma \in \Sigma^\omega$ *and* $\forall i \in \mathbb{N} \cdot \sigma_{i,i} \Vdash D_{\langle * \rangle}$.

Proof. Given a general i then σ_i is of the form $\langle E \rangle$, and by definition of $D_{\langle * \rangle}$ and by the first rule of Defn. 2.1 then $(q_0, \theta_0) \overset{E}{\underset{(}{\Rightarrow}} q_F, \theta_0)$. By Defn. **??** then $\sigma_i \Vdash D_{\langle * \rangle}$ follows immediately.

B Proofs for Section 3 (Monitors as Triggers for LTL Formulas)

Proposition 3. $\sigma_{i,j} \vdash D; \varphi \implies \forall k > j \cdot \sigma_{i,k} \nvdash D; \varphi$.[11]

Proof. ($\implies$) If $\sigma_{i,j} \vdash D; \varphi$ then $\sigma_{i,j}$ can be divided into $\sigma_{i,j'}$ and $\sigma_{j',j}$ such that $\sigma_{i,j'} \Vdash D$ and $\sigma_{j',j} \Vdash \varphi$ is also true.

An easy corollary of Proposition 1 is that there if a trace flags then any strict prefix of it does not flag. Then j' is unique here. Furthermore, Definition 2 ensures that $\sigma_{j',j}$ does not a strict prefix that also satisfies φ.

Consider for contradiction that $\exists k > j \cdot \sigma_{i,k} \vdash D; \varphi$. Then we know that $\sigma_{i,j'} \Vdash D$ and $\sigma_{j',k} \Vdash \varphi$ (as determined before the choice of j' here is the only choice). Definition 2 here causes a contradiction, since $\sigma_{j',j} \Vdash \varphi$ implies that $\sigma_{j',j} \vdash \varphi$, but $\sigma_{j',k} \Vdash \varphi$ implies that $\sigma_{j',j} \nvdash \varphi$. □

Proposition 4. $\sigma \vdash \phi \iff \sigma \vdash D_{\langle * \rangle} : \phi$.[12]

Proof. ($\implies$) By definition of $\sigma_{o,\infty} \vdash D_{\langle * \rangle} : \phi$, Definition 3, then $\exists j \cdot 0 \leq j \wedge (\sigma_{0,j} \Vdash D_{\langle * \rangle} \implies \sigma_{j,\infty} \vdash \phi)$. Since $D_{\langle * \rangle}$ flags upon only one event, by Proposition 2, then $\sigma_{0,j} \Vdash D_{\langle * \rangle}$ is true for $j = 0$. But $\sigma_{0,\infty} \vdash \phi)$ is equivalent to $\sigma \vdash \phi$, which we assumed. □

($\impliedby$) Suppose $\sigma \vdash D_{\langle * \rangle} : \phi$. Then $\sigma_0 \Vdash D_{\langle * \rangle}$, from Proposition 2. Then by the first rule of Definition 3 $\sigma_{0,\infty} \vdash \phi$, which is equivalent to the left-hand side. □

C Proofs for Section 4 (Synthesising Monitor-Triggered Controllers)

Theorem 3. *The formula $t(\pi) = \gamma \rightarrow \varphi$ is tightly realisable iff it is realisable. A Mealy machine tightly realising $t(\pi)$ can be constructed from $C_{t(\pi)}$ with the same complexity.*

Proof. Clearly, if $t(\pi)$ is not realisable then it cannot be tightly realisable.

Suppose that $t(\pi)$ is realisable and let $C_{t(\pi)}$ be the Mealy machine realising it. We show how to augment $C_{t(\pi)}$ with accepting states.

First, given a co-safety formula φ, we can construct a deterministic finite automaton accepting all *finite* prefixes u such that $u_{0,|u|} \vdash \varphi$. Construct an

[11] Proof can be found in Appendix B.
[12] Proof can be found in Appendix B.

alternating weak automaton (AWW) A_φ for φ using standard techniques [16]. The only states of this automaton that have self loops are states of the form $\psi_1 U \psi_2$. It follows that the only option for this automaton to accept is by a transition to $\mathbf{tt}$. We construct a nondeterministic finite automaton (NFW) N_φ from A_φ by using a version of the subset construction and having the empty set as the only accepting state of the NFW. We then construct a deterministic finite automaton D_φ by applying the classical subset construction to N_φ.

The AWW A_φ is linear in φ, the NFW N_φ is at most exponential in φ, and the DFW D_φ is at most doubly exponential in φ.

We then take the product of D_φ with the Mealy machine $C_{t(\pi)}$.

As $C_{t(\pi)}$ realises $t(\pi)$, whenever a word w satisfies γ it must satisfy φ as well. Then, as D_φ accepts all prefixes that satisfy (tightly) φ, the first prefix of w that satisfies φ is accepted by the Mealy machine.[13]

Note that the construction identifying violating prefixes for a safety language [17] would not produce the required result: the automaton constructed using that technique would identify the empty trace as satisfying $X\mathbf{tt}$. □

Theorem 4. *Let $C_{t(\pi)}$ be a Mealy machine realising $t(\pi)$ when π' is a simple-trigger LTL, and tightly realising $t(\pi)$ when π' is a repeating-trigger LTL. Then there is a Mealy machine $M \blacktriangleright C_{t(\pi)}$ that realises π.*

Proof. We give a construction for such a controller, and prove it realises π.

The monitor-activated controller $M \blacktriangleright C$ is defined as the Mealy machine with the following components $\langle S, s_0, \mathcal{P}_{in}, \mathcal{P}_{out}, \rightarrow, \emptyset \rangle$, where $S = (Q_M \times \Theta_M) \cup Q_C$, $s_0 = (q_M^0, \theta_0)$, $\mathcal{P}_{in}$ and $\mathcal{P}_{out}$ are shared among all three. The transition relation $\rightarrow$ is the minimal relation respecting:

1. If the monitor is in a state that has an outgoing transition to a non-final state then take that transition. Note that this also applies to sink states.

$$\frac{q'_M \notin F \qquad (q_M, \theta) \xrightarrow{I}_M (q'_M, \theta')}{(q_M, \theta) \xrightarrow{I/\emptyset} (q'_M, \theta')}$$

2. If the monitor is in a state that has an outgoing transition to a final state then take that transition synchronously with a corresponding transition from the controller initial state.

$$\frac{q'_M \in F_M \qquad (q_M, \theta) \xrightarrow{I}_M (q'_M, \theta') \qquad q_C^0 \xrightarrow{I/O}_C q_C}{(q_M, \theta) \xrightarrow{I/O} q_C}$$

3. If the controller C is in a non-accepting state take transitions from that state.

$$\frac{q_C \notin F_C \qquad q_C \xrightarrow{I/O}_C q'_C}{q_C \xrightarrow{I/O} q'_C}$$

[13] We conjecture that realisability procedures relying on determinisation and the Mealy machines constructed from them could be further analysed to produce the required tight Mealy machine without requiring the additional automaton D_φ.

4. If the controller C has a transition to an accepting state then transition to the initial state of the monitor. Note that this will not be activated in the case of simple triggers, for which we will be using controllers without accepting states.

$$\frac{q'_C \in F_C \qquad q_C \xrightarrow{I/O} q'_C}{q_C \xrightarrow{I/O} (q^0_M, \theta^0)}$$

To prove this construction realises π, we consider two cases, when π' is a simple-trigger formula $D{:}\phi$ and when it is a repeating-trigger formula $(D;\varphi)^*$. Throughout we assume we are considering traces that satisfy the assumption, otherwise the specification is trivially satisfied.

In the case of a simple trigger, it is easy to see that there are two cases: either the monitor never flags, or it flags. If the monitor never flags (i.e. only rule 1 is ever used), then π is trivially satisfied. If the monitor flags, then a suffix of the trace is produced by $C_{t(\pi)}$ (note rules 2 and 3). That is, there is a j such that $\sigma_{0,j} \Vdash D$ and $\sigma_{j,\infty}$ is produced by $C_{t(\pi)}$. Note that rule 4 in the construction is never activated, since $C_{t(\pi)}$ will not have accepting states. However every trace produced by $C_{t(\pi)}$ also satisfies $t(\pi)$, since the former realises the latter . The result then easily follows.

In the case of a repeating trigger, there are two cases: either the monitor flags finitely often, or the monitor flags infinitely often.
Suppose the monitor flags finitely often. Consider by induction that the monitor flags zero times, then it is easy to see that the trace satisfies $(D;\varphi)^*$, since D never flags. The inductive step, when the monitor flags $n+1$ times, can be easily reduced to nth case by pruning from the trace the prefix that satisfies $D;\varphi$. This prefix must exist since the monitor must flag at least once and upon entering $C_{t(\pi)}$, which tightly realises the co-safety formula $t(\pi)$ and thus will accept.
Suppose the monitor flags infinitely often, and for contradiction suppose the formula is not realised. There must then be some trace whose prefix does not satisfy $D;\varphi$. This prefix in turn must have a prefix that satisfies D, otherwise the whole specification $D;\varphi$ is satisfied. However, since D must flag on this violating trace then, since $C_{t(\pi)}$ tightly realises the co-safety formula $t(\pi)$, we are guaranteed the prefix will eventually be extended to accept and satisfy φ. Thus $D;\varphi$ must be satisfied by the prefix, which creates a contradiction. □