Monitoring Realizability*

Rüdiger Ehlers and Bernd Finkbeiner

Reactive Systems Group
Saarland University
66123 Saarbrücken, Germany

Abstract. We present a new multi-valued monitoring approach for linear-time temporal logic that classifies trace prefixes not only according to the existence of correct and erroneous continuations, but also according to the strategic power of the system and its environment to avoid or enforce a violation of the specification. We classify the monitoring status into four levels: (1) the worst case is a *violation*, where no continuation satisfies the specification any more; (2) *unrealizable* means that the environment can force the system to violate the specification; (3) *realizable* means that the system can enforce that the specification is satisfied; (4) the best case, *fulfilled*, indicates that all possible continuations satisfy the specification. Because our approach recognizes situations where the system cannot avoid a violation even though there may still be continuations in which the specification is satisfied, our approach detects errors earlier, and it detects errors that are missed by less detailed classifications. We give an asymptotically optimal construction of multi-valued monitoring automata based on parity games.

1 Introduction

One of the guiding principles of runtime monitoring is that violations of the specification should be reported *as early as possible*, giving the user (or controller) time to act before the violation causes serious harm. The principle means that the monitor must reason about the future: we issue a warning as soon as we can *predict* that a violation is about to occur. The standard implementation of this idea is to consider a finite trace as *bad* if all its infinite extensions violate the specification. In other words, as long as there *exists* a future in which the specification is satisfied, we assume that this future will actually occur and do not issue a warning.

In this paper, we revisit this optimistic interpretation of the future. In reality, not all future actions are under the system's control. It is therefore possible to reach situations where the system can no longer *avoid* the violation, even though there exists some continuation in which the violation does not occur. Such situations are important early indicators of failure: we know for sure that

* This work was partly supported by the German Research Foundation (DFG) under the project SpAGAT (grant no. FI 936/2-1) in the priority programme "Reliably Secure Software Systems – RS3".

ignite	○	●	○	○	○	○	●	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○ ···
charge	○	○	●	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○ ···
spark	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	●	○	○	○	●	○	○	○ ···
Monitor	□	□	□	□	□	□	□	⋆	⋆	⋆	⋆	⋆	□	□	□	□	□	□	□	↯	↯	↯	↯ ···

Legend:
○ *false*
● *true*
✓ *fulfilled*
□ *realizable*
⋆ *unrealizable*
↯ *violation*

Fig. 1. Example execution and monitoring trace of a faulty ignition controller. The controller responds correctly to the first ignition request, by charging the coil and then emitting a spark. The specification is realizable (□), but not fulfilled (✓), because there exists a continuation that violates the specfication. After the second ignition request, the controller fails to charge the coil. The monitor therefore switches from *realizable* to *unrealizable* (⋆). At the third ignition request, the monitor switches back to *realizable* and stays, because the controller responds correctly, in *realizable* until the spontaneous third spark occurs. At this point, the monitor recognizes a *violation* (↯).

the system does not satisfy its specification, we just cannot guarantee that the violation will be visible on the execution we are about to see.

Consider a car ignition controller that needs to charge a coil before emitting a spark, and that is, to save energy, only allowed to start charging when an ignition request is issued. The controller has an input signal *ignite* and two output signals *charge* and *spark*, whose behavior could be specified in linear-time temporal logic as follows (abstracting from implementation details like the charging time and other activities of the ignition controller):

$$\begin{aligned}\psi = \neg \mathit{spark} \;\wedge\; \neg \mathit{spark}\,\mathcal{W}\,\mathit{charge} \;\wedge \Box(\mathit{spark} \rightarrow \bigcirc\,(\neg \mathit{spark}\,\mathcal{W}\,\mathit{charge}))\\ \wedge\, \neg \mathit{charge} \;\wedge\; \Box(\bigcirc\,\mathit{charge} \rightarrow \mathit{ignite}) \;\wedge\; \Box(\mathit{ignite} \rightarrow \bigcirc\Diamond\,\mathit{spark})\end{aligned}$$

Under what circumstances should we raise an alarm? Clearly, it is appropriate to issue a warning if *charge* is activated without an ignition request or if a spark is emitted without previously charging the coil. In both situations, the specification is definitely violated, because there is no possible continuation into the future that would satisfy the specification. However, a smart observer would be able to recognize problems earlier than that. Suppose an ignition request is given, but the system does not *immediately* charge the coil in the next step. It is easy to see that this is a mistake, because now the system can no longer prevent the conjunct $\Box(\mathit{ignite} \rightarrow \bigcirc\Diamond\,\mathit{spark})$ from becoming false: if no more ignition requests come in, then the coil will never be charged, and, hence, the spark can never be emitted.

How can we recognize such mistakes? One way to characterize the situation is to observe that there exists an extension that violates the specification. This condition is easy to check, we simply monitor the negation of the specification as well. However, issuing a warning whenever there exists a violating extension would be overly pessimistic: Since, right from the start, there always exists a path that violates the specification, we would continuously warn that things "may" go wrong.

The exact right time for the warning is when there is an ignition request but the coil is not charged in the very next step. At this point, we not only

know that there exists a continuation that violates the specification, we can actually identify the inputs a malicious environment would need to produce in order to *enforce* that the violation will occur. We say that in this situation the specification is *unrealizable*. While it is still too early to tell if the specification will really become violated (if a second ignition comes in and the system charges the coil at that time, things are fine), the system under observation must be faulty because there is no system that satisfies the specification for all possible future inputs.

In this paper, we give a precise analysis of the possible futures by distinguishing the different roles played by *outputs*, which are under the system's control, and *inputs*, which are chosen by the (potentially hostile) environment. This results in a finer classification of the monitoring situation with four different conditions, going from worst case to best case as follows:

1. *violation:* the specification is definitely violated, i.e., there is no more continuation that satisfies the specification;
2. *unrealizable:* the specification is not violated but unrealizable, i.e., the environment can force the system to violate the specification;
3. *realizable:* the specification is not fulfilled but realizable, i.e., there is a continuation in which the specification is violated, but the system can enforce that the specification is satisfied; and
4. *fulfilled:* the specification is definitely satisfied, i.e., there is no continuation that violates the specification.

Figure 1 shows an execution trace of a faulty ignition controller, which occasionally fails to charge the coil and at some point produces a spontaneous spark. The monitor starts out in condition *realizable*, and switches to *unrealizable* when there is an ignition request but the coil is not charged. This alert is serious: a bug has been detected. However, the monitor does not report a violation yet, and indeed, in the trace, the user reacts by requesting another ignition, and when, this time, the coil is charged, the monitor switches back to *realizable*. Only when, later, there is a spontaneous spark, the monitor raises the alarm: the specification is definitely violated at that point.

Semantically, our approach is a departure from the classic linear-time approach to runtime monitoring. Distinguishing inputs and outputs naturally leads to *games*, rather than sets of traces, as the underlying model of computation. Figure 2 shows the game between the ignition controller and its environment. The two players take turns. In the states owned by the system player the ignition controller chooses the outputs, in the states owned by the environment player, the environment chooses the inputs. The winning condition is expressed as a parity condition: if the highest number that appears infinitely often during a play of the game is even, then the system player wins, otherwise the environment player wins. From the initial state, the system player has a winning strategy: always stay in states A through F. If the system player deviates from this strategy by moving from state C to state G, then the environment player has a winning strategy: from state G, always move to state H, never back to C. If, however, the environment player does at some point move back to C, then

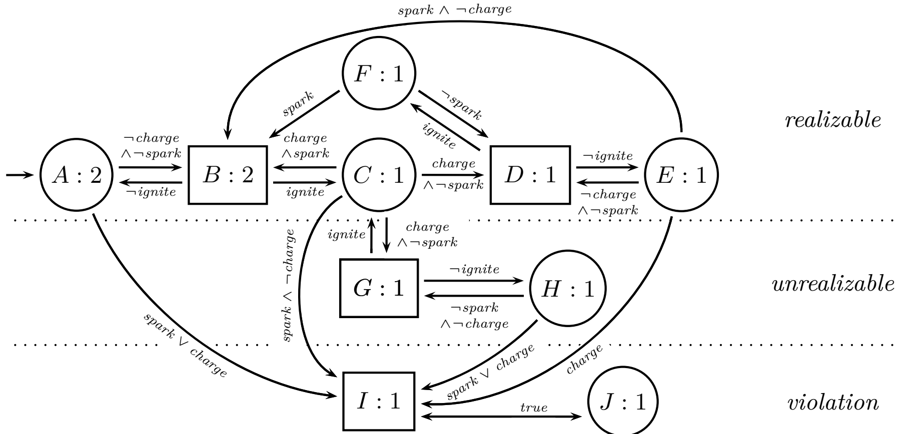

Fig. 2. Parity game between the ignition controller and its environment. Positions owned by the system player are shown as circles, positions owned by the environment as squares. The system player has a winning strategy from states A through F, the environment player has a winning strategy in all other states. From states I and J, the environment player wins no matter how the strategy is chosen. A runtime monitor tracing this game will report *realizable* in states A through F, *unrealizable* in states G and H, and *violation* in states I and J.

the system player has again a winning strategy. The game is definitely lost for the system player if the play reaches states I or J, indicating that the system player has issued a spark or charge out of turn. From these states, the game is won by the environment player, no matter which moves are chosen.

The runtime monitor traces the states in the game while processing the observations from the monitored system. In states A through F, the status is *realizable*, because the system player has a winning strategy, in states G and H, the status is *unrealizable*, because the environment player has a winning strategy, and in states I and J, the status is *violation*, because the environment player wins independently of the strategy. In the paper, we explain the construction of the game and the resulting monitor in more detail. We start by converting the specification to an equivalent deterministic parity automaton and its corresponding parity game. Solving the game partitions the automaton into sets of states corresponding to the four monitoring conditions. Based on this classification, we construct a finite-state machine that implements the monitor.

In the last technical section of the paper, Section 4, we add one more twist to the game-based analysis: in addition to recognizing whether one of the players has a winning strategy, we check if the violation or fulfillment of the specification can be enforced *in a finite number of steps*. This allows the user to estimate the urgency of the *unrealizable* monitoring status: if the number of steps is finite, then the system is in imminent danger; if not, we know that, while the system cannot avoid the violation without help from the environment, the system can at least *delay* the violation for an unbounded number of steps.

Related work. There has been a long debate in runtime verification about the best way to translate specifications, which refer to infinite computations, into monitors, which are limited to observing finite prefixes. Kupferman and Vardi coined the term *informative prefix* for prefixes that "tell the whole story" why a specification is violated [11]. The advantage of informative prefixes is that one can monitor the specification without analyzing the future. For example, one can translate the specification into a small equivalent alternating automaton and track the active states in disjunctive or conjunctive normal form [7]. However, informative prefixes are usually longer than necessary. For example, an informative prefix of the specification $\bigcirc\, \mathit{false}$ has length one, although one could deduce the violation of the formula without seeing any trace at all. In order to recognize violations earlier, one needs to quantify over the possible futures. A prefix is *bad* [11] if there is no infinite extension that satisfies the specification. In order to construct a monitor that recognizes the bad prefixes, one translates the formula into an equivalent nondeterministic Büchi automaton, eliminates all states with empty language, and then determinizes with a powerset construction into an automaton on finite words that recognizes the bad prefixes. d'Amorim and Roşu showed that the runtime overhead caused by monitoring can be reduced significantly by recognizing when the observed prefix can no longer be extended to a bad prefix and pruning such "Never-Violate" states from the monitor [3].

Our approach to check *realizability* in addition to satisfiability builds on algorithms for reactive synthesis. In synthesis, we check whether the specification is realizable, i.e., whether there exists an implementation for the given specification. Similar to our monitoring approach, one analyzes the game between the system and its environment and searches for a winning strategy for the system player [2]. The key difference between checking and monitoring realizability is, however, that in synthesis we only check for the existence of a strategy from the initial state, whereas in monitoring we make this judgment again and again, as we observe a growing prefix of a trace.

The monitoring work that is closest to our approach is *interface monitoring* as proposed by Pnueli et al. [16]. In this work, an interface monitor is compiled from a module implementation together with its interface specification. The analysis considers a game, where the nondeterminism of the module is seen as one player and the interface behavior as the other. It is assumed that the interface is trying to satisfy both its own specification and the global specification, and the module is trying to produce a violation. In contrast to this approach, we monitor the behavior of the system rather than its interface, because we are interested in execution faults where the behavior of the system deviates from its specification. In order to obtain monitors of reasonable size, we also avoid encoding the implementation of any part of the system into the monitor.

The approach of this paper can be seen as an extension of three-valued monitoring of linear-time temporal logic [1]. Taking the input/output interface of a system into account significantly increases the usefulness of multi-valued monitoring, because now even violations of liveness constraints that depend on input to the system can be detected.

2 Monitoring Reactive Systems

We are interested in monitoring *reactive systems*, which interact with their environment over a potentially infinite run. We start by recalling standard notions and constructions from runtime monitoring.

2.1 Preliminaries

Interfaces. The *interface* of a reactive system is defined as a tuple $\mathcal{I} = (\mathrm{AP}_I, \mathrm{AP}_O)$, where AP_I is a finite set of input signals to the system and AP_O is a finite set of output signals. Together, the two sets form the *atomic propositions* $\mathrm{AP} = \mathrm{AP}_I \uplus \mathrm{AP}_O$ of the system. During the execution of the system, it produces a (potentially infinite) word $w = w_0 w_1 \ldots$, where, in every step, the valuation of the input signals is read and the respective valuation of the output signals is produced, i.e., for every $i \in \mathbb{N}$, $w_i \in 2^{\mathrm{AP}_I} \times 2^{\mathrm{AP}_O}$. We call the words produced by the execution of a system also the *traces* of the system. Depending on whether it is assumed that in every step first the input or output is read, the system model corresponds to the one of Mealy or Moore machines [13], respectively. The techniques in this paper are equally applicable in both models, although we assume a Moore machine model in the following.

Execution trees. The behavior of a (deterministic) reactive system with interface $\mathcal{I} = (\mathrm{AP}_I, \mathrm{AP}_O)$ can be represented as an infinite tree $\langle T, \tau \rangle$, where $T \subseteq (2^{\mathrm{AP}_I})^*$ is the set of nodes of the tree, and $\tau : T \rightarrow 2^{\mathrm{AP}_O}$ is the labeling function of the tree, i.e., it decorates every node of the tree with an output. The meaning of an execution tree is as follows. If $t = t_0 \ldots t_n \in T$ is the input of the system read since the system went into service, then $\tau(t)$ is the output of the system in the $n+1$st clock cycle. We say that an infinite *path* $p = p_0 p_1 \ldots \in (2^{\mathrm{AP}_I})^\omega$ *induces* a word/trace $w = (p_0, \tau(\epsilon))(p_1, \tau(p_0))(p_2, \tau(p_0 p_1)) \ldots \in (2^{\mathrm{AP}_I} \times 2^{\mathrm{AP}_O})^\omega$ in the execution tree. An execution tree is called *full* if $T = (2^{\mathrm{AP}_I})^*$.

The idea behind execution trees is that the decision of the next output is based on the entire history of inputs received so far. A reactive system is assumed to have a full execution tree: because it has no control over the input, any input sequence can arise during its execution.

We say that an execution tree (or a reactive system represented by the tree) satisfies some word language $L \subseteq (2^{\mathrm{AP}_I} \times 2^{\mathrm{AP}_O})^\omega$ if every word that is induced by some path in the tree is contained in L.

Linear-time temporal logic (LTL). LTL [14] is a commonly used specification logic for reactive systems. LTL describes linear-time properties, i.e., sets of correct traces. Formulas in LTL are built from atomic propositions, Boolean operators and the temporal operators $\Box$ (globally), $\Diamond$ (finally), $\mathcal{U}$ (until) and $\mathcal{W}$ (weak until). Given an infinite trace $w = w_0 w_1 \ldots \in (2^{\mathrm{AP}})^\omega$ over some set of atomic propositions AP, we define the satisfaction of an LTL formula inductively over the structure of the LTL formula. Let ϕ_1 and ϕ_2 be LTL formulas and w^i denote the suffix of a word $w = w_0 w_1 \ldots$ starting from the ith element, i.e., $w^i = w_i w_{i+1} \ldots$. The semantics of LTL is defined as follows:

- $w \models p$ if and only if (iff) $p \in w_0$ for $p \in \mathrm{AP}$
- $w \models \neg\psi$ iff not $w \models \psi$
- $w \models (\phi_1 \vee \phi_2)$ iff $w \models \phi_1$ or $w \models \phi_2$
- $w \models (\phi_1 \wedge \phi_2)$ iff $w \models \phi_1$ and $w \models \phi_2$
- $w \models \bigcirc\phi_1$ iff $w^1 \models \phi_1$
- $w \models \Box\phi_1$ iff for all $i \in \mathbb{N}$, $w^i \models \phi_1$
- $w \models \Diamond\phi_1$ iff there exists some $i \in \mathbb{N}$ such that $w^i \models \phi_1$
- $w \models (\phi_1 \mathcal{U} \phi_2)$ iff there exists some $i \in \mathbb{N}$ such that for all $0 \leq j < i$, $w^j \models \phi_1$ and $w^i \models \phi_2$
- $w \models (\phi_1 \mathcal{W} \phi_2)$ iff for every $i \in \mathbb{N}$ such that $w^0 \not\models \phi_2$, $w^1 \not\models \phi_2$, ..., $w^{i-1} \not\models \phi_2$ and $w^i \not\models \phi_2$, also for all $0 \leq j < i$, $w^j \models \phi_1$.

The set of traces that satisfy an LTL formula is called its *language*. The *length* of an LTL formula is defined as the number of occurrences of operators and atomic propositions. We say that an execution tree (or a reactive system) satisfies an LTL formula ψ if it satisfies the language of the formula.

Runtime monitoring. As discussed under related work, there are multiple definitions of the LTL runtime monitoring problem. The "standard" problem defined in the following is based on three-valued monitoring [1]. We wish to observe the trace of the reactive system and raise an *alarm* whenever the trace prefix cannot be completed into an infinite trace that satisfies the specification, and to raise a *success* signal whenever the trace cannot be completed to one that does not satisfy the specification. Given an LTL formula ϕ over a set of atomic propositions AP, we can build a *monitor automaton* for ϕ, i.e., a finite state machine that observes the input and output of a system and where every state is labeled by *safe*, *unknown* or *bad*. During the run of the monitor, the state labels represent whether the prefix trace observed witnesses the violation or satisfaction of the formula by every continuation of the prefix trace. Formally, such a monitor is represented as a tuple $\mathcal{M} = (S, \Sigma, \delta, s_0, L)$, where S is the set of states, $\Sigma = 2^{\mathrm{AP}}$ is the input alphabet, $\delta : S \times \Sigma \to S$ is the transition function, $s_0 \in S$ the initial state and $L : S \to \{\mathit{safe}, \mathit{unknown}, \mathit{bad}\}$ is the labeling function. We also say that (S, Σ, δ, s_0) is the *transition structure* of $\mathcal{M}$. Given a finite word $w = w_0 w_1 \ldots w_n \in (2^{\mathrm{AP}})^n$, we say that w induces a (prefix) run $\pi = \pi_0 \ldots \pi_{n+1}$ in $\mathcal{M}$ such that $\pi_0 = s_0$ and for every $i \in \{0, \ldots, n\}$, we have $\pi_{i+1} = \delta(\pi_i, w_i)$. By abuse of notation, we write $\pi_{n+1} = \delta(s_0, w_0 \ldots w_n)$.

A finite-state machine $\mathcal{M} = (S, \Sigma, \delta, s_0, L)$ with $\Sigma = 2^{\mathrm{AP}}$ represents a monitor for an LTL formula ϕ over AP if the following conditions are satisfied: (1) for every $w \in (2^{\mathrm{AP}})^*$, $L(\delta(s_0, w)) = \mathit{bad}$ if and only if for all $w' \in (2^{\mathrm{AP}})^\omega$, $ww' \not\models \phi$ (so the formula can no longer be satisfied); and (2) for every $w \in (2^{\mathrm{AP}})^*$, $L(\delta(s_0, w)) = \mathit{good}$ if and only if for all $w' \in (2^{\mathrm{AP}})^\omega$, $ww' \models \phi$ (so the formula will be satisfied whatever happens in the future). We call the set of prefix traces that lead to a good state in a monitor the *good prefixes*, and the prefix traces that lead to a bad state in a monitor its *bad prefixes*.

Constructing runtime monitors for LTL. There are standard constructions to translate LTL formulas to monitor automata. For reference, we quickly

recall the construction described in [1]. We start by building nondeterministic automata for both the specification and its negation. In these automata, we prune states with empty language and then determinize with a powerset construction. The product of the resulting deterministic finite-word automaton represents a monitor with a doubly-exponential number of states in the length of the original specification. As shown by Kupferman and Vardi [11], there is a doubly-exponential lower bound and the construction is therefore essentially optimal[1].

2.2 Constructing monitors from deterministic parity automata

In preparation for our main construction in Section 3, which is based on parity games, we now present an alternative monitor construction via deterministic parity automata.

A *deterministic parity automaton* is a tuple $\mathcal{A} = (Q, \Sigma, \delta, q_0, c)$ with the set of states Q, the alphabet Σ, the transition function $\delta : Q \times \Sigma \rightarrow Q$, the initial state q_0 and the coloring function $c : Q \rightarrow \mathbb{N}$. Given an infinite word $w = w_0 w_1 \ldots$, w induces a run $\pi = \pi_0 \pi_1 \ldots$ over $\mathcal{A}$, where $\pi_0 = q_0$ and for every $i \in \mathbb{N}$, $\pi_{i+1} = \delta(\pi_i, w_i)$. Likewise, a finite word $w = w_0 w_1 \ldots w_n$ induces a finite run $\pi = \pi_0 \pi_1 \ldots \pi_{n+1}$ in $\mathcal{A}$ where $\pi_0 = q_0$ and for every $i \in \{0, \ldots, n\}$, we have $\pi_{i+1} = \delta(\pi_i, w_i)$. We say that an infinite word w is in the language of $\mathcal{A}$, denoted by $\mathcal{L}(\mathcal{A})$, if and only if for the run $\pi = \pi_0 \pi_1 \ldots$, the highest number occurring infinitely often in the sequence $c(\pi_0), c(\pi_1), c(\pi_2), \ldots$ is even. For the scope of this paper we require, without loss of generality, the transition function to be a complete function. We refer to (Q, Σ, δ, q_0) as the *transition structure* of $\mathcal{A}$.

Given an LTL formula ϕ over a set of atomic propositions AP, we can translate ϕ to a deterministic parity automaton $\mathcal{A}$ over the alphabet 2^{AP} such that for every infinite word $w \in (2^{\mathrm{AP}})^\omega$, we have $w \models \phi$ if and only if $w \in \mathcal{L}(\mathcal{A})$. The automaton $\mathcal{A}$ has $2^{O(2^n n \log n)}$ states and $3(n+1)2^n$ colors [18].

In order to build a monitor for an LTL formula from its equivalent deterministic parity automaton, we need to identify the states with universal or empty language. Given an automaton $\mathcal{A} = (Q, \Sigma, \delta, q_0, c)$, for every $q \in Q$, we denote by $\mathcal{A}_q$ the automaton $(Q, \Sigma, \delta, q, c)$, i.e., the same automaton but with a different initial state. If for a $q \in Q$, $\mathcal{L}(\mathcal{A}_q) = \emptyset$, we say that q has an *empty language*, or if $\mathcal{L}(\mathcal{A}_q) = \Sigma^\omega$, we say that q has *universal language*. To identify the states with empty language, we check each of the automata $\mathcal{A}_q$ for $q \in Q$ for emptiness (see [5] for a suitable procedure). States with universal language are identified by doing the same on a version of the automaton where 1 is added to every color, which complements the language of each state. Based on the sets of states with the empty and universal language, we identify bad and good prefixes:

---

[1] Kupferman and Vardi prove a $2^{2^{\Omega(\sqrt{n})}}$ lower bound, while the construction from [1] leads to an automaton of size 2^{2^n}, where n denotes the length of the LTL formula. The difference is negligible, however, because we can carry out a precise finite-state machine minimization [8] after the construction of the monitor.

Lemma 1. *Let $\mathcal{A} = (Q, 2^{\mathrm{AP}}, \delta, q_0, c)$ be a deterministic parity automaton that is obtained by a translation from an LTL formula ψ over the set of atomic propositions* AP*, $E \subseteq Q$ be the set of the states of $\mathcal{A}$ that have an empty language, and $U \subseteq Q$ be the set of states of $\mathcal{A}$ that have a universal language.*

For every finite word $w \in (2^{\mathrm{AP}})^*$, w induces a run in $\mathcal{A}$ that ends in a state in E iff w is a bad prefix for ψ. Likewise, w induces a run in $\mathcal{A}$ that ends in a state in U iff w is a good prefix for ψ.

With this lemma, we can now transform the deterministic parity automaton into a monitor: we take the same set of states, label every state with an empty parity automaton language with *bad* and every state with a universal parity automaton language with *good*.

The monitor based on the parity automaton is slightly larger than the one described in the previous subsection ($2^{O(2^n n \log n)}$ states compared to 2^{2^n} states)[2]. The advantage of using the transition structure of the deterministic parity automaton is, however, that it allows us to recognize realizability, as we will see in the following section.

3 Monitoring Realizability

As discussed in the introduction, a monitor that only detects bad and good prefixes misses early indicators of failure, where the environment can enforce a violation of the specification. Such a violation of *realizability* means that the system under observation is incorrect, because there exists an input that will cause a violation of the specification, but the situation is less severe than the occurrence of a bad prefix, because the bad input might not actually occur during the current run of the system.

3.1 Parity games

In a parity game, two players play for an infinite duration of time. The game consists of a set of states, which are connected by labeled edges. Every state is assigned to one of the two players, *Player 0* and *Player 1*. The game is played by moving a pebble along the edges of the game. Whenever the pebble is on a state that belongs to the some player, this player gets to choose the action. The pebble then moves according to the edge function to a state of the opposing player. Every state has a color. A play is won by Player 0 if the highest color visited infinitely often along the play is even.

Formally, a *parity game* is a tuple $\mathcal{G} = (V_0, V_1, \Sigma_0, \Sigma_1, E_0, E_1, v_{in}, c)$. $V = V_0 \uplus V_1$ are the states, where the states in V_0 belong to Player 0 and the states in V_1 belong to Player 1. Σ_0 and Σ_1 are the action sets, $E_0 : V_0 \times \Sigma_0 \to V_1$ and $E_1 : V_1 \times \Sigma_1 \to V_0$ are the edge functions of the two players. Additionally, $v_{in} \in V$ is the initial state and $c : V \to \mathbb{N}$ is the coloring function.

[2] We can apply precise finite-state machine minimization [8] after the construction to obtain a monitor of equal size.

A *decision sequence* in $\mathcal{G}$ is a sequence $\rho = \rho_0^0 \rho_0^1 \rho_1^0 \rho_1^1 \ldots$ such that for all $i \in \mathbb{N}$, $\rho_i^0 \in \Sigma_0$ and $\rho_i^1 \in \Sigma_1$. A decision sequence ρ induces an infinite *play* $\pi = \pi_0^0 \pi_0^1 \pi_1^0 \pi_1^1 \ldots$ if $\pi_0^0 = v_0$ and for all $i \in \mathbb{N}$, $p \in \{0, 1\}$, $E_p(\pi_i^p, \rho_i^p) = \pi_{i+p}^{1-p}$.

Given a play $\pi = \pi_0^0 \pi_0^1 \pi_1^0 \pi_1^1 \ldots$, we say that π is winning for Player 0 if $\max\{c(v) \mid v \in V_0, v \in \inf(\pi_0^0 \pi_1^0 \ldots)\}$ is even, where the function inf maps a sequence to the set of elements that appear infinitely often in the sequence. If a play is not winning for Player 0, it is winning for Player 1.

Given some parity game $\mathcal{G} = (V_0, V_1, \Sigma_0, \Sigma_1, E_0, E_1, v_0, \mathcal{F})$, a strategy for Player 0 is a function $f : (\Sigma_0 \times \Sigma_1)^* \to \Sigma_0$. Likewise, a strategy for Player 1 is a function $f : (\Sigma_0 \times \Sigma_1)^* \times \Sigma_0 \to \Sigma_1$. In both cases, a strategy maps prefix decision sequences to an action to be chosen next. A decision sequence $\rho = \rho_0^0 \rho_0^1 \rho_1^0 \rho_1^1 \ldots$ is said to be *in correspondence with* f if for every $i \in \mathbb{N}$, we have $\rho_n^p = f(\rho_0^0 \rho_0^1 \ldots \rho_{n+p-1}^{1-p})$. A strategy is *winning* for Player p if all plays in the game that are induced by some decision sequence that is in correspondence to f are winning for Player p.

Parity games are *determined*, which means that there exists a winning strategy for precisely one of the players. We call a state $v \in V$ winning for player p if the player has a winning strategy in the modified game where the initial state has been changed to v.

Parity games and reactive systems. Parity games are a common model for the interaction of a system with its environment. Player 0 represents the system, Player 1 the environment. Player 0's actions thus consist of the outputs, Player 1's actions of the inputs.

We can translate a given LTL formula into a parity game such that there is an execution tree that satisfies the formula along all its words if and only if there exists a winning strategy for Player 0 from the initial state. Given a winning strategy f, we can build a suitable execution tree $\langle T, \tau \rangle$ by taking the decisions of the system player as the tree labels: $T = (2^{\mathrm{AP}_I})^*$ and $\tau(t_0 \ldots t_n) = f((\tau(\epsilon), t_0)(\tau(t_0), t_0 t_1)(\tau(t_0 t_1), t_0 t_1 t_2) \ldots (\tau(t_0 \ldots t_{n-1}), t_0 \ldots t_n))$ for every $t_0 \ldots t_n \in T$.

Definition 1. *Given a deterministic parity automaton $\mathcal{A} = (Q, \Sigma, \delta, q_0, c)$ with $\Sigma = 2^{\mathrm{AP}_I} \times 2^{\mathrm{AP}_O}$, we build its induced parity game $\mathcal{G} = (Q, Q \times 2^{\mathrm{AP}_O}, 2^{\mathrm{AP}_O}, 2^{\mathrm{AP}_I}, E_0, E_1, q_0, c')$ with*

$$\forall v_0 \in Q, x_0 \in \Sigma_0, E_0(v_0, x_0) = (v_0, x_0);$$
$$\forall v_0 \in Q, x_0 \in \Sigma_0, x_1 \in \Sigma_1, E_1((v_0, x_0), x_1) = \delta(v_0, (x_1, x_0));$$
$$\forall v_0 \in Q, x_0 \in \Sigma_0, c'(v_0) = c(v_0) \text{ and } c'((v_0, x_0)) = 0.$$

Lemma 2. *Given a deterministic parity automaton $\mathcal{A} = (Q, \Sigma, \delta, q_0, c)$ with $\Sigma = 2^{\mathrm{AP}_I} \times 2^{\mathrm{AP}_O}$, there exists a winning strategy for the system player from the initial state of the game induced by $\mathcal{A}$ iff there exists an execution tree for the interface $(\mathrm{AP}_I, \mathrm{AP}_O)$ for which all induced words are in $\mathcal{L}(\mathcal{A})$.*

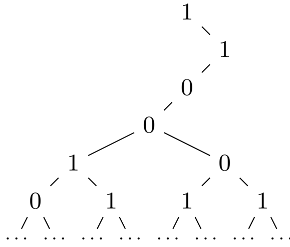

Fig. 3. Example bobble tree over $\mathrm{AP}_I = \{i\}$ and $\mathrm{AP}_O = \{o\}$. The tree branches according to 2^{AP_I}, where the left children correspond to $i = 0$ and the right children correspond to $i = 1$. The tree nodes are labelled by the value of o. The tree has the split word $\{i, o\}\{o\}\emptyset$ and describes the past behaviour of a reactive system with interface $\mathcal{I} = (\mathrm{AP}_I, \mathrm{AP}_O)$ after having read $\{i\}\emptyset\emptyset$ from its initial state. The tree branches according to all possible inputs from the split node onwards.

3.2 Recognizing realizability

We now formalize the situations in which the monitor should report *realizable* and *unrealizable*. We call prefixes that lead to a realizable situation *winning* and prefixes that lead to an unrealizable situation *losing*, corresponding to the intuition that, in a realizable situation, Player 0 has a winning strategy, and in an unrealizable situation, all strategies of Player 0 lose. The formal definition is based on the concept of bobble trees, which are a special case of execution trees: Bobble trees combine the representation of the *past* of an execution, which is a prefix trace, with the representation of the *future*, which is a full tree.

A *bobble tree* $\langle T, \tau \rangle$ has a split node $\overline{t} = \overline{t}_0 \dots \overline{t}_n \in T$ such that for every node $t \in T$ either t is a prefix of $\overline{t}$, or $\overline{t}$ is a prefix of t and furthermore $\overline{t}t' \in T$ for every $t' \in 2^{\mathrm{AP}_I}$. Thus, the tree has a single unique path to the split node $\overline{t}$ and is full only from that point onwards. We call the prefix word $w = (\tau(\epsilon), \overline{t}_0)(\tau(\overline{t}_0), \overline{t}_1) \dots (\tau(\overline{t}_0 \dots \overline{t}_{n-1}), \overline{t}_n)$ the *split word* of $\langle T, \tau \rangle$. Figure 3 shows an example of a bobble tree.

Definition 2. *Let $\mathcal{I} = (\mathrm{AP}_I, \mathrm{AP}_O)$ be an interface and $L \subseteq (2^{\mathrm{AP}_I} \times 2^{\mathrm{AP}_O})^\omega$ be a language. We say that some prefix word $w = w_0 \dots w_n \in (2^{\mathrm{AP}_I} \times 2^{\mathrm{AP}_O})^*$ is a winning prefix (for L) if there exists some bobble tree with split word w that satisfies L. Likewise, we say that some prefix word $w = w_0 \dots w_n \in (2^{\mathrm{AP}_I} \times 2^{\mathrm{AP}_O})^*$ is a losing prefix if all bobble trees with split word w do not satisfy L.*

It is easy to see that bad prefixes are special cases of losing prefixes, and dually, good prefixes are special cases of winning prefixes. The following theorem forms the basis of our approach for monitoring for winning and losing prefixes:

Theorem 1. *Let $\mathcal{A} = (Q, \Sigma, \delta, q_0, c)$ be a deterministic parity automaton with $\Sigma = 2^{\mathrm{AP}_I} \times 2^{\mathrm{AP}_O}$ and $\mathcal{G}$ be the corresponding parity game. For every prefix word $w = w_0 \dots w_n \in \Sigma^*$ with its associated path $\pi = \pi_0 \dots \pi_{n+1}$ in $\mathcal{A}$, w is a winning/losing prefix for $\mathcal{L}(\mathcal{A})$ iff π_{n+1} is a state in $\mathcal{G}$ that is winning/losing for Player 0, respectively.*

Proof. Assume that w is a winning prefix. This is equivalent to the fact that there exists a tree $\langle T, \tau \rangle$ where there is no prefix word other than w of length $|w|$ and from the node $w|_I = (w_0 \cap \mathrm{AP}_I)(w_1 \cap \mathrm{AP}_I) \ldots (w_n \cap \mathrm{AP}_I)$ onwards, the tree is full and all of its paths are in the language of $\mathcal{A}$. This is the case if and only if, from π_{n+1} onwards, all words in the the sub-tree from node $w|_I$ are accepted by $\mathcal{A}_{\pi_{n+1}}$. By the definition of $\mathcal{G}$, this in turn is equivalent to π_{n+1} being winning for Player 0. The argument for losing prefixes is dual. □

We have thus connected the monitoring problem for reactive systems to parity game solving. Since parity games are determined (i.e., every state is winning for precisely one of the two players), we directly obtain as a corollary:

Corollary 1. *Let $\mathcal{A} = (Q, \Sigma, \delta, q_0, c)$ be a deterministic parity automaton with $\Sigma = 2^{\mathrm{AP}_I} \times 2^{\mathrm{AP}_O}$. Every finite word $w \in \Sigma^*$ is either a winning or a losing prefix.*

A monitor can therefore only encounter the following four situations: *fulfilled* if the prefix is good, *realizable* if the prefix is winning but not good, *unrealizable* if the prefix is losing but not bad, and *violation* if the prefix is bad.

We construct the monitor by identifying which states in the deterministic parity automaton are winning for Player 0 in the respective game, and combine the information with the information about states in the automaton witnessing *good* and *bad* prefixes. The monitor has the same transition structure as the parity automaton. In terms of complexity, we obtain the following:

Theorem 2. *Let $\mathcal{I} = (\mathrm{AP}_I, \mathrm{AP}_O)$ be an interface and ψ be an LTL formula over $\mathrm{AP}_I \uplus \mathrm{AP}_O$. Building a finite-state machine that distinguishes between bad (and losing), losing, winning and good (and winning) prefixes is 2EXPTIME-complete.*

Proof. For the lower bound, we note that the 2EXPTIME-complete [15] problem of checking the realizability of LTL formulas is a special case: If we synthesize a monitor, we can easily check for the realizability of a specification by testing whether the initial state of the monitor machine is labeled by *good* or *winning*.

For the upper bound, we start by building an automaton $\mathcal{A}$ with $2^{O(2^n n \log n)}$ states and $3(n+1)2^n$ colors [18] that is equivalent to ψ, where n is the length of ψ. Dividing the set of states in the corresponding game into the winning ones and the losing ones can be done in time $2^{|\mathrm{AP}_I|+|\mathrm{AP}_O|} m^{O(d)}$ [9], where m is the number of states in the game (i.e., $m = (2^{|\mathrm{AP}_O|} + 1) \cdot 2^{O(2^n n \log n)}$) and d is the number of colors in the game (i.e., $d = 3(n+1)2^n$). Combined with the effort to identify the monitor states that represent good and bad prefixes, we obtain a doubly-exponential time bound for this procedure. □

4 Finitary Winning And Losing Prefixes

We now add a further refinement to the classification of monitoring situations: we distinguish situations in which the system or environment can enforce fulfillment

or violation, respectively, in *finite* time. The extended classification provides helpful information about the urgency of the problem behind the *unrealizable* status. Suppose, for example, that the monitor of a flight control system informs an airplane pilot that the environment of the control system can force a violation of the specification in finite time. Since a violation of the specification is imminent, the pilot might take drastic action in such a case, such as perform an emergency landing. If, on the other hand, the environment needs infinite time to enforce a violation, there is much more time for diagnosis and decision. It may well be a better idea to continue the flight and report the system malfunction (or incorrect specification) after the regular landing.

In this section, we define *finitary winning* and *finitary losing* prefixes and show how to adapt the monitor construction from the previous section to also detect these. Using this addition, our monitors for reactive systems now have six monitoring conditions, going from worst case to best case as follows:

1. *violation:* the prefix is bad;
2. *unrealizable with finite time:* the prefix is finitary losing but not bad;
3. *unrealizable with infinite time:* the prefix is losing but not finitary losing;
4. *realizable with infinite time:* the prefix is winning but not finitary winning;
5. *realizable with finite time:* the prefix is finitary winning but not good; and
6. *fulfilled:* the prefix is good.

We begin by formalizing the definition of finitary losing and winning prefixes.

Definition 3. *Let $\mathcal{A} = (Q, \Sigma, \delta, q_0, c)$ be a deterministic parity automaton with $\Sigma = 2^{\mathrm{AP}_I} \times 2^{\mathrm{AP}_O}$. We say that some prefix word $w = w_0 \dots w_n \in (2^{\mathrm{AP}_I} \times 2^{\mathrm{AP}_O})^*$ is a* finitary winning prefix *if there exists some bobble tree $\langle T, \tau \rangle$ with split word w such that every infinite word in $\langle T, \tau \rangle$ has a good prefix word. Likewise, we say that some prefix word $w = w_0 \dots w_n \in (2^{\mathrm{AP}_I} \times 2^{\mathrm{AP}_O})^*$ is a* finitary losing prefix *if for all bobble trees $\langle T, \tau \rangle$ with split word w, there exists an infinite word in $\langle T, \tau \rangle$ that has a bad prefix word.*

The following lemma characterizes the finitary winning and losing prefixes in terms of the parity game, which allows us to base the monitors for such prefixes on the framework described in the previous sections.

Lemma 3. *Let $\mathcal{A} = (Q, \Sigma, \delta, q_0, c)$ be a deterministic parity automaton with $\Sigma = 2^{\mathrm{AP}_I} \times 2^{\mathrm{AP}_O}$, $E \subseteq Q$ be the states of $\mathcal{A}$ that have an empty language, $U \subseteq Q$ be the set of states of $\mathcal{A}$ that have a universal language and $\mathcal{G}$ be the game corresponding to $\mathcal{A}$.*

- *For every prefix word $w = w_0 \dots w_n \in \Sigma^*$, w is a* finitary winning prefix *iff for the corresponding prefix run $\pi = \pi_0 \dots \pi_{n+1}$, Player 0 has a strategy from state π_{n+1} to eventually visit U.*
- *For every prefix word $w = w_0 \dots w_n \in \Sigma^*$, w is a* finitary losing prefix *iff for the corresponding prefix run $\pi = \pi_0 \dots \pi_{n+1}$, Player 1 has a strategy from state π_{n+1} to eventually visit E.*

As a consequence, we can again use the transition structure of the deterministic parity automaton for our monitor. The only addition to the previous monitor construction is that we need to identify the states in the game which allow Player 0 and Player 1 to force the play into one of the states whose corresponding state in the parity automaton has a universal or empty language, respectively. For this purpose, we apply a standard *attractor* [12] construction on the game graph. To compute the finitary winning states, we initialize the attractor with the states U whose language is universal and then repeatedly add states owned by Player 0 that have an outgoing edge into the attractor, and states owned by Player 1 where all outgoing edges lead into the attractor. The fixpoint of this construction contains exactly those states where Player 0 can force the game into U in a finite number of states. Analogously, we compute the finitary losing states with an attractor that is initialized with the states E whose language is empty, and where we repeatedly add states owned by Player 1 with an edge to the attractor and states owned by Player 0 where all edges lead to the attractor. The computation of the attractor sets takes linear time in the size of the game [12]. We obtain as a corollary:

Corollary 2. *Let $\mathcal{I} = (\mathrm{AP}_I, \mathrm{AP}_O)$ be an interface and ψ be an LTL formula over $\mathrm{AP}_I \uplus \mathrm{AP}_O$. Building a finite-state machine that distinguishes between bad, finitary losing, losing, winning, finitary winning, and good prefixes is 2EXPTIME-complete.*

5 Conclusion

We have presented a new multi-valued monitoring approach for linear-time temporal logic that classifies trace prefixes not only according to the correctness of the continuations, but also according to the strategic power available to the system and its environment in order to avoid or enforce a violation. The game-based approach has several advantages over the classic approaches: the game-based analysis detects errors earlier, it detects errors that are missed by purely trace-based approaches, and it can indicate the urgency with which a violation is to be expected.

Our constructions are optimal in the complexity-theoretic sense. A potential drawback of our approach is that we construct a deterministic automaton. Other monitoring techniques construct nondeterministic or universal automata, which are, in theory, exponentially more compact. The determinization is then often done symbolically, for example in hardware using individual flip-flops for the states of the nondeterministic or universal automaton (cf. [6]).

However, experiments with state-of-the-art LTL-to-automata translators have shown that nondeterministic automata are not necessarily smaller than deterministic automata. For many practical specifications, the deterministic automaton is in fact smaller than the nondeterministic automaton originally produced by the translator [10, 4]. Constructing deterministic automata and applying an efficient symbolic encoder [17] may thus even lead to smaller, faster and more memory-efficient monitors.

References

1. Bauer, A., Leucker, M., Schallhart, C.: Runtime verification for LTL and TLTL. ACM Transactions on Software Engineering and Methodology **20**(4) (2011)
2. Büchi, J., Landweber, L.: Solving sequential conditions by finite-state strategies. Trans. AMS (138) (1969)
3. d'Amorim, M., Rosu, G.: Efficient monitoring of ω-languages. In Etessami, K., Rajamani, S.K., eds.: CAV. Volume 3576 of LNCS., Springer (2005) 364–378
4. Ehlers, R.: Minimising deterministic Büchi automata precisely using SAT solving. In Strichman, O., Szeider, S., eds.: SAT. Volume 6175 of LNCS., Springer (2010) 326–332
5. Ehlers, R.: Short witnesses and accepting lassos in ω-automata. In Dediu, A.H., Fernau, H., Martín-Vide, C., eds.: LATA. Volume 6031 of LNCS., Springer (2010) 261–272
6. Finkbeiner, B., Kuhtz, L.: Monitor circuits for LTL with bounded and unbounded future. In Bensalem, S., Peled, D., eds.: RV. Volume 5779 of LNCS., Springer (2009) 60–75
7. Finkbeiner, B., Sipma, H.: Checking finite traces using alternating automata. Formal Methods in System Design **24**(2) (2004) 101–127
8. Hopcroft, J.E.: An n log n algorithm for minimizing the states in a finite automaton. In Kohavi, Z., ed.: The Theory of Machines and Computations, Academic Press (1971) 189–196
9. Jurdzinski, M.: Small progress measures for solving parity games. In Reichel, H., Tison, S., eds.: STACS. Volume 1770 of LNCS., Springer (2000) 290–301
10. Klein, J., Baier, C.: Experiments with deterministic ω-automata for formulas of linear temporal logic. Theor. Comput. Sci. **363**(2) (2006) 182–195
11. Kupferman, O., Vardi, M.Y.: Model checking of safety properties. Formal Methods in System Design **19**(3) (2001) 291–314
12. Küsters, R.: Memoryless determinacy of parity games. In Grädel, E., Thomas, W., Wilke, T., eds.: Automata, Logics, and Infinite Games. Volume 2500 of LNCS., Springer (2001) 95–106
13. Müller, S.M., Paul, W.J.: Computer architecture: complexity and correctness. Springer (2000)
14. Pnueli, A.: The temporal logic of programs. In: FOCS, IEEE (1977) 46–57
15. Pnueli, A., Rosner, R.: On the synthesis of an asynchronous reactive module. In Ausiello, G., Dezani-Ciancaglini, M., Rocca, S.R.D., eds.: ICALP. Volume 372 of LNCS., Springer (1989) 652–671
16. Pnueli, A., Zaks, A., Zuck, L.D.: Monitoring interfaces for faults. Electr. Notes Theor. Comput. Sci. **144**(4) (2006) 73–89
17. Sentovich, E., Singh, K., Lavagno, L., Moon, C., Murgai, R., Saldanha, A., Savoj, H., Stephan, P., Brayton, R.K., Sangiovanni-Vincentelli, A.L.: SIS: A system for sequential circuit synthesis. Technical Report UCB/ERL M92/41, EECS Department, University of California, Berkeley (1992)
18. Vardi, M.Y., Wilke, T.: Automata: from logics to algorithms. In Flum, J., Grädel, E., Wilke, T., eds.: Logic and Automata: History and Perspectives. Number 2 in Texts in Logic and Games. Amsterdam University Press (2007) 629–736