Monitor-Based Runtime Assurance for Temporal Logic Specifications

Matthew Abate, Eric Feron, and Samuel Coogan

***Abstract*— This paper introduces the *safety controller* architecture as a runtime assurance mechanism for system specifications expressed as safety properties in Linear Temporal Logic. The safety controller uses a *monitor*, constructed as a finite state machine, to analyze a desired control input policy online and form a sequence of control inputs that is guaranteed to keep the system safe for all time. A case study is presented which details the construction and implementation of a safety controller on a cyber-physical system with a nondeterministic dynamical model.**

I. Introduction

Modern cyber-physical systems (CPS) are complex and sometimes do not behave as expected. Correctness can be partially addressed with offline verification, however, the end-to-end verification of an entire system is often prevented by its complexity. In order to compensate for the lack of assurances, it is desirable to enforce correctness online.

For purely cyber systems, runtime correctness can be checked online using monitors [1], [2], [3], [4], [5]. In this context, correctness is evaluated with respect to a temporal logic specification; a monitor observes the temporal behaviors of the cyber system and notifies the system operator if a fault is suspected. Such monitors are convenient for verification because they can be algorithmically generated from temporal logic specifications; however, monitors can only detect system faults and do not have the ability to enforce correct behavior at runtime. As an alternative, an edit automaton can be employed in the control loop to assure correct system behavior online [6]. Edit automata are formed corresponding to temporal logic specifications, however, it is unclear how hard it is to generate an edit automaton given such a specification. It was shown in [7], for instance, that an edit automaton for a given specification is not unique.

For controlled dynamical systems, runtime correctness can be enforced online; typically, this procedure involves ensuring that the system does not exceed the boundary of a known controlled forward-invariant region in the state-space. Numerous verification techniques exist in this paradigm, including level set based methods [8], barrier certificate based methods [9], and certain model predictive control (MPC) based methods [10], [11]. These approaches, however, are typically only well suited to analyze invariance specifications or reach specifications; more complex temporal behaviors cannot be enforced in this manner.

This research was supported in part by the National Science Foundation under award #1749357.

An extended version of this work is availible on ArXiVe, `https://arxiv.org/abs/1908.03284`

M. Abate is with the School of Mechanical Engineering, Georgia Institute of Technology, Atlanta, 30332, USA `Matt.Abate@GaTech.edu`.

E. Feron is with the School of Aerospace Engineering, Georgia Institute of Technology, Atlanta, 30332, USA `Eric.Feron@Aerospace.GaTech.edu`.

S. Coogan is with the School of Electrical and Computer Engineering and the School of Civil and Environmental Engineering, Georgia Institute of Technology, Atlanta, 30332, USA `Sam.Coogan@GaTech.edu`.

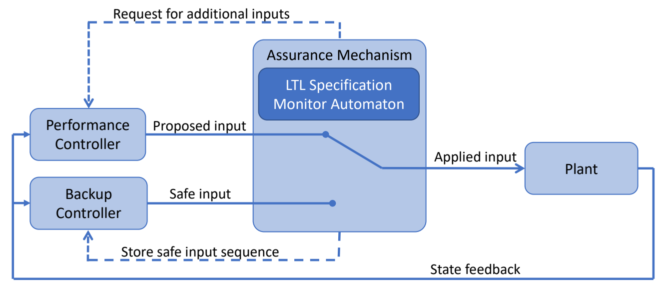


Fig. 1: Safety Controller Architecture

It is not overtly clear how one might combine results for purely cyber systems with results for physical systems in order to ensure correct behavior of CPS against complex specifications. For example, by the time a monitor recognizes that a physical system is about to violate its operating specification, the system may have entered an invariant region of the state space from which it is impossible to satisfy the system specification going forward. A basic idea that has emerged in different contexts is to enforce correct performance online through the incorporation of a backup control policy. This technique is referred to as "runtime assurance" or the Simplex Architecture [12].

This paper introduce the *safety controller*: a correct-by-construction runtime assurance mechanism system for mission objectives encoded in linear temporal logic (LTL) . The safety controller (Figure 1) has three fundamental components: a performance controller, a backup controller and an assurance mechanism. The assurance mechanism uses a monitor, constructed as a finite-state machine (FSM), to assess the performance control input and generate a provably correct control input policy at runtime. In this setting, the backup controller is characterized by a subset of the system-monitor statespace where correct performance can be assured on an infinite time horizon; importantly, the proposed framework allows the performance controller to steer the system outside this region, provided that the performance controller can demonstrate safety beforehand. This creates a trade-off between the *a priori* computation required to generate a backup controller and the real-time computational burden placed on the safety controller.

We organize this paper in the following way. We introduce monitor automata, a correct-by-construction tool for runtime verification, in Section II. Section III presents the problem formulation and then introduces the safety controller architecture. We provide a discussion on the trade-off between offline platform development and online assurance in Section IV, and the paper concludes with an experimental demonstration, provided in Section V.

II. Preliminaries on Monitoring for Linear Temporal Logic Properties

In this section, we introduce *monitor automata* as a runtime verification tool from automata based model checking.

A. Monitor Construction in LTL$_3$

To formally describe the execution of a CPS, we encode system events as *atomic propositions*; a sequence of events, or a *word*, therefore denotes the progression of the system through time. We use AP to denote a set of atomic propositions, $\Sigma = 2^{AP}$ to denote a finite alphabet, and Σ^* and Σ^ω to denote the sets of finite and infinite system traces over Σ, respectively. System properties are specified in Linear Temporal Logic (LTL). For an in depth discussion on the semantics of LTL, we refer the reader to [13] Section 2.1.

Consider an LTL specification φ and a finite path fragment $w \in \Sigma^*$. Note that there may be no continuations of w which satisfy φ, or equivalently, the system may have reached a state where it can no longer satisfy its specification under any possible future execution. We therefore introduce the definitions of good and bad prefixes in order to describe the set of finite words from which it is impossible to violate or satisfy the mission objective.

Definition 1 (Good and Bad Prefixes)**.** Consider a finite word $w_1 \in \Sigma^*$ and an LTL property φ. w_1 is said to be a *good prefix* for φ if $w_1 w_2 \models \varphi$ for all $w_2 \in \Sigma^\omega$. Similarly, w_1 is said to be a *bad prefix* for φ if $w_1 w_2 \not\models \varphi$ for all $w_2 \in \Sigma^\omega$.

In order to classify whether a finite word is a good or bad prefix for φ, we use the *truth value* of the mission objective.

Definition 2 (LTL$_3$ Semantics, [2] Section 2.2)**.** Let $w \in \Sigma^*$ denote a finite word. The *truth value* of an LTL$_3$ formula φ with respect to w, denoted $[w \models \varphi]$, is an element of $\mathbb{B}_3 = \{\top, \bot, ?\}$ defined as follows:

$$[w \models \varphi] = \begin{cases} \top & \text{if } \forall \sigma \in \Sigma^\omega \ : \ w\sigma \models \varphi \\ \bot & \text{if } \forall \sigma \in \Sigma^\omega \ : \ w\sigma \not\models \varphi \\ ? & \text{otherwise.} \end{cases}$$

Equivalently, the truth value φ with respect to w is *true* "$\top$" if w is a good prefix for φ, *false* "$\bot$" if w is a bad prefix for φ, and *inconclusive* "$?$" otherwise.

Finally, we formalize the automata-based monitoring procedure for LTL$_3$ (Definition 3). We refer the reader to [2] for a comprehensive discussion on monitor automata, and we provide a sample monitor construction in Figure 2.

Definition 3 (Monitors in LTL$_3$)**.** For a given property φ a *monitor automaton* $\mathcal{M}^\varphi$ is a finite state machine that reads finite words $w \in \Sigma^*$ and outputs $[w \models \varphi]$.

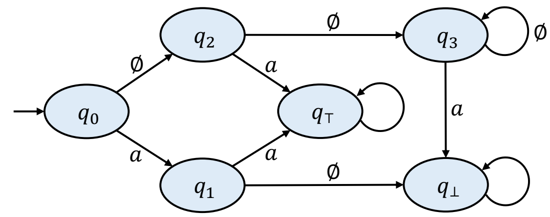


Fig. 2: Monitor Automaton $\mathcal{M}^\varphi$ where $\varphi = \Box\varnothing \vee \bigcirc a$ is an LTL property evaluated over $\Sigma = \{\varnothing, a\}$. Monitor states $q_\top$ and $q_\bot$ output *true* and *false*, respectively, and $q_0, \cdots, q_3$ output *inconclusive*.

Note that, for a given mission objective, any continuation of a good/bad prefix for that property will also be a good/bad prefix. Consequently, a monitor $\mathcal{M}^\varphi$ will never have more than one state with output $\top$ and one state with output $\bot$ after minimization. In the following, we use the symbols $q_\top$ and $q_\bot$ to denote monitor states with outputs $\top$ and $\bot$, respectively.

B. Monitoring Safety Properties

In the following, we focus the scope of our study to LTL properties which guarantee certain monitor output behaviors. Specifically, we recall the definition of *safety properties* [14].

Definition 4 (Safety Property)**.** An LTL property φ_s is said to be a *safety property* if all words which violate φ_s contain a bad prefix for φ_s.

LTL safety properties make up a broad class of system specifications; for example, $\varphi = \Box\varnothing \vee \bigcirc a$ (Figure 2) is a safety property in LTL. Furthermore, applying Definition 2.3, we can guarantee system safety by ensuring that the system trace does not contain a bad prefix for φ_s. We formalize this result in Proposition 1.

Proposition 1. *An infinite word $w \in \Sigma^\omega$ satisfies φ_s if $[\hat{w} \models \varphi_s] \neq \bot$ for all prefixes $\hat{w} \in Pref(w)$.*

Note that a monitor $\mathcal{M}^{\varphi_s}$, for a safety property φ_s not semantically equal to *true*, is guaranteed to have a state with output $\bot$. Moreover, we can verify system trajectories against φ_s by ensuring the system run over $\mathcal{M}^{\varphi_s}$ never enters $q_\bot$.

III. The Safety Controller Architecture

In this section, we apply monitor automata in a framework which enforces system objectives at runtime. For system objectives expressed as safety properties in LTL, we create a notion of system safety and describe a runtime assurance mechanism which enforces safe system behavior.

A. Assurance Through Monitoring

We model CPS as discrete-time non-deterministic control systems of the form

$$x^+ = f(x, u, d) \tag{1}$$

where $x_k \in \mathcal{X} \subseteq \mathbb{R}^n$, $u_k \in \mathcal{U} \subseteq \mathbb{R}^m$ and $d_k \in \mathcal{D} \subset \mathbb{R}^p$ represent the system state, the control input and a non-deterministic bounded disturbance at a time $k \in \mathbb{N}_{\geq 0}$, respectively.

Associated with the system (1) is a set of atomic propositions AP and a labeling function $L : \mathcal{X} \to 2^{AP}$. For the remainder of this paper, we use the notation $L(x_1, \cdots, x_n)$ to denote the string of labels $L(x_1, \cdots, x_n) := L(x_1) \cdots L(x_n)$, and we use the symbols Σ^* and Σ^ω to denote the sets of finite and infinite words over Σ, respectively. Additionally, let φ be property in LTL. We aim to ensure that a safety property $\varphi \in$ LTL is satisfied over a (infinite) system run of (1).

Following Proposition 1, we say that a finite system trajectory $x_0, \cdots, x_k$ is *safe* if $[L(x_0 \cdots x_k) \models \varphi] \in \{\top, ?\}$, and *unsafe* if $[L(x_0 \cdots x_k) \models \varphi] = \bot$. A controller which is implemented on the system must first ensure the system stays safe before pursuing any auxiliary system objectives, such as enforcing optimality constraints or attempting to satisfy additional LTL properties.

B. Components of a Safety Controller

In the following, we assume knowledge of a controller which is claimed to be able to keep the system safe, while also possibly meeting some auxiliary system objective. This controller, denoted a *performance controller*, is characterized by three assumptions:

Assumption III.1. At each time k, the performance controller proposes a control input to be applied to the system.

Assumption III.2. If requested, the performance controller can also propose a sequence $(X_k, g_k), \cdots, (X_{k+N_{max}}, g_{k+N_{max}})$ of feedback control laws $g_i : \mathcal{X} \to \mathcal{U}$ and regions of the state space $X_i \subseteq \mathcal{X}$, such that at a future time i, the performance controller intends to choose its inputs using g_i provided the current state of the system $x_i \in X_i$. The performance controller is not obligated to choose future inputs using these policies.

Assumption III.3. The performance controller might bear faulty, i.e. applying the performance control input to (1) may eventually cause the system to violate its specification φ.

This definition of the performance controller is quite broad and encompasses general unverified feedback control laws. For example, a human operator, whose effectiveness cannot be verified *a priori*, can be thought of as a performance controller for a manned CPS. Despite the lack of global assurances, we assume that certain aspects of the performance controller law make it preferable; for instance, the performance controller may be designed to optimize some objective or achieve some auxiliary goal.

We additionally assume knowledge of a controller which is known to satisfy the safety objective, but might be limited in abilities or performance. We denote this mechanism a *backup controller*, formalized next.

Definition 5 (Backup Controller and High Assurance Region)**.** For a system of the form (1), a safety property $\varphi \in$ LTL, and a corresponding monitor $\mathcal{M}^\varphi = (\Sigma, Q, q_0, \delta, \lambda)$, let $S = \mathcal{X} \times Q$ denote the total state space of the combined system and monitor. A *backup controller* is characterized by a subset of the state space $S^b \subseteq S$ such that for any $(x, q) \in S^b$, there exists an infinite sequence of control inputs known to the backup controller such that the resulting infinite horizon system trace satisfies φ. S^b is referred to as the *high assurance region* of the backup controller.

For the remainder of this section we fix a safety property φ, a corresponding monitor $\mathcal{M}^\varphi = (\Sigma, Q, q_0, \delta, \lambda)$, a backup controller and a high assurance region S^b. Trivially, the system can be kept safe for all time by applying the backup control input to the system at each time step, however, this method of input generation restricts the system to operating inside the high assurance region and removes the possibility of reaching auxiliary goals. For this reason, we interpret the problem formulation as follows: create a logical architecture which chooses whether to apply the performance control input or the backup control input to the system at each time step such that the resulting infinite-time system trace satisfies φ. In this case, whenever possible, the chosen input should be that of the performance controller.

In the following, we refer to the aforementioned logical architecture as the *assurance mechanism* of the safety controller. Importantly, the assurance mechanism should allow the system to leave S^b if a control policy is known which returns the system to S^b at a future time. We justify this decision with Proposition 2.

Proposition 2. *Let $x_0, \cdots, x_k$ denote a finite trajectory of system (1), and let $q_0, \cdots, q_k$ be the corresponding run of the trajectory over $\mathcal{M}^\varphi$. If $(x_k, q_k) \in S^b$, then $L(x_0 \cdots x_k)$ is not a bad prefix for φ.*

It follows from Proposition 2 that if the performance controller is able to suggest a path back to the high assurance region S^b from every point along a system trajectory, then the performance controller is functioning correctly. In order to check this condition, we call Algorithm 1.

Algorithm 1 takes an initial state $(x, q) \in \mathcal{X} \times Q$, and returns a finite sequence of control inputs γ_r such that:

1) $\gamma_r = \varnothing$ signifies a fault in the performance controller, and
2) $\gamma_r = g_0, \cdots, g_k$ signifies that choosing inputs according to the sequence of feedback control laws sequential will return the system to S^b.

This procedure relies, in part, on the calculation of finite time reachable sets, notated as follows. For an initial state $(x_0, q_0) \in S$, and a sequence of feedback control laws $g_0, g_1, \cdots$, we denote the set of reachable states after i steps by R_i, i.e.

$$\begin{aligned} R_1 = \{(x, q) \mid x = f(x_0, g_0(x_0), d), \\ d \in \mathcal{D},\ q = \delta(q_0, L(x))\} \\ R_{i+1} = \{(x, q) \mid x = f(\bar{x}, g_i(\bar{x}), d),\ (\bar{x}, \bar{q}) \in R_i, \\ d \in \mathcal{D},\ q = \delta(\bar{q}, L(x))\} \end{aligned}$$

for all $i > 1$. While calculating reachable sets can be computationally expensive, numerous methods exist to approximate reachable sets efficiently. Therefore, let $\tilde{R}_i$ denote an over-approximation of R_i, i.e. $R_i \subseteq \tilde{R}_i$.

We check to see whether the performance controller is functional by calling RECOVERY(x, q), provided in Algorithm 1. Succinctly, RECOVERY(x, q) simulates the system dynamics using the current system monitor state and a non-empty sequence of suggested future performance control inputs; if the future system monitor state is contained in S^b, then the suggested input sequence is returned, and, if not, the null sequence is returned, signaling a system fault. We use the term *recovery input sequence* to denote a sequence of control inputs which drive the current system state into S^b. Therefore, if the suggested control input is guaranteed to drive the system to S^b, then the output of Algorithm 1 is a non-empty recovery input sequence for the current system monitor state.

Algorithm 1 Generate A Non-Empty Recovery Input Sequence in the Presence of Disturbances

input : a current state $(x, q) \in S$.
output: a sequence of feedback control laws γ_r
function RECOVERY(x, q)
  Initialize: $\tilde{R}_0 = \{(x, q)\}$, $i = 0$
  while $(i \leq N_{max})$ **do**
    $X_i, g_i \leftarrow$ Request_Next_Control_Law
    if $\{x \mid (x, q) \in \tilde{R}_i\} \nsubseteq X_i$ **then**
      return $\gamma_r = \varnothing$
    Compute $\tilde{R}_{i+1}$ using $\tilde{R}_i$ and g_i
    if $(\tilde{R}_{i+1} \subseteq S^b)$ **then**
      return $\gamma_r = g_0, \cdots, g_i$
    $i \leftarrow i + 1$
  return $\gamma_r = \varnothing$
end function

We next use Algorithm 1 to design a logical architecture which chooses between the performance control input and the backup control input at each timestep. This procedure is implemented through the following steps:

1) At each time k the assurance mechanism calls RECOVERY(x_k, q_k) and stores the output in a variable γ_r.
2) If γ_r is non-empty, then $g_0(x_k)$ is applied to the system and $g_1, \cdots, g_j$ is stored to memory.
3) If $\gamma_r = \varnothing$, then the performance controller has suggested an input sequence that cannot be verified. The memorized recovery input sequence is applied to return the system to S^b, and the backup control input is then applied for all future time.

This procedure is implemented with Algorithm 2. Importantly, choosing control inputs by Algorithm 2 guarantees system safety for all time; see Theorem 1.

Algorithm 2 Runtime Assurance for Non-Deterministic Discrete-Time Control Systems

Initialize: $x = x_0$, $q = q_0$
$\gamma_r \leftarrow$ RECOVERY(x, q)
while $(\gamma_r \neq \varnothing)$ **do**
  Apply $g_0(x)$ to system;
  Store $g_1, \cdots, g_j$ to **Memory**
  $(x, q) \leftarrow$ CURRENT_STATE
  $\gamma_r \leftarrow$ RECOVERY(x, q)
Apply Recovery Input Sequence to system
while (**True**) **do**
  Apply BACKUP_INPUT to system

Theorem 1 (Runtime Assurance for Non-Deterministic CPS)**.** *Let $\varphi \in LTL$ be a safety property, and let $w = L(x_0\, x_1 \cdots)$ be the trace of an infinite trajectory resulting from a sequence of inputs chosen using Algorithm 4. If the system is initialized in the high assurance region, i.e. $(x_0, q_0) \in S^b$, then $w \models \varphi$.*

The proof of this result is available in an extended version of this paper: see `https://arxiv.org/abs/1908.03284`.

IV. CLASSIFYING THE TRADE-OFF BETWEEN A PRIORI DEVELOPMENT AND EFFECTIVENESS

Here, we discuss the feasibility of designing and implementing a safety controller architecture. Specifically, we address the offline computation involved in constructing a backup controller and the computational complexity of the algorithms which run online in the assurance mechanism.

A. Developing a Safety Controller Offline

Two steps are required to construct a safety controller from an LTL safety specification φ and a given system model: the first step is to convert the system specification to a monitor automaton and the next step is to design a backup controller.

We refer the reader to [2] for a detailed discussion on monitor construction. In short, this process involves realizing two deterministic finite automata (DFAs) and then computing their minimal product automaton. There are well-defined procedures for constructing a DFA from an LTL specification, as well as procedures for computing product automata [15]. The resulting product automaton is reduced to its minimal form by removing every unreachable state and every pair of non-distinguishable states. This procedure is implemented using Moore's algorithm [16], which has an average complexity $O(n \cdot \log(n))$ when minimizing a DFA with n states.

A backup control law is developed by identifying a controlled invariant region of the system monitor state space $S^b \subseteq \mathcal{X} \times Q$, such that for all $(x, q) \in S^b$, $q \neq q_\perp$. In order to compute such an invariant region, abstraction based methods are possible [13], [17].

B. Online Computation for the Assurance Mechanism

We next attempt to characterize the online computational resources required by a safety controller. First, we propose that Assumption III.2, which requires the performance controller to suggest potential future inputs, fits well with existing control architectures; for instance, all MPC controllers employ this functionality. Moreover, as the performance controller is assumed to be provided in advance of the development process, we do not discuss the computational complexity of designing a performance controller.

Next, we note that the assurance mechanism must have sufficient computational capabilities to over-approximate reachable system states at runtime. Reachable set computation is a well-studied problem in the controls and hybrid systems literature, with numerous efficient algorithms. See [18, Chapter 29] for an overview.

C. Discussion

The assurance mechanism will only choose the performance control input if the set of reachable states resulting from that sequence is contained inside the high assurance region. There is, therefore, an intuitive trade-off between the *a priori* development of the assurance mechanism and the amount of computational resources which are necessary online. For example, the likelihood of finding a safe performance control input sequence is maximized for safety controllers with large, well developed, high assurance regions. Similarly, an assurance mechanism which computes tight approximations of reachable sets will be less likely to return a fault flag, in comparison to an assurance mechanism which uses conservative approximations.

There are in fact other methods of increasing the capabilities of the safety controller, beyond those presented previously. For instance, note that if the performance controller is ever determined to be faulty, Algorithm 2 first applies the recovery input sequence, driving the system to reenter S^b, and then applies the backup control input for all future time; an assurance mechanism may instead choose to reactivate the performance controller once S^b is reached. In this case, the performance controller is certainly faulty and may again suggest an unsafe sequence of inputs at some-point in the future. However, even in this instance, the resulting infinite-time system trace is guaranteed to satisfy the mission objective. As a second extension, we suggest that if the system leaves S^b, with assurance from the performance controller, then the assurance mechanism might choose to expand S^b on the fly with the knowledge of the new system state and its corresponding recovery input sequence. As the S^b expands, the system will effectively *learn* how to stay safe in previously unverified product states, increasing system performance. Safe learning is an active research area in the controls and formal methods communities. One modern technique, referred to as shielding, enforces LTL safety properties in a runtime assurance framework [19]. Current shielding methods, however, only assure discrete-time discrete-state systems, whereas the safety controller architecture is applicable to systems with a continuous state space. Shielding also assumes a probabilistic system disturbance, which we intentionally avoid in our construction.

V. Case Study: An Accelerating DeLorean

To demonstrate the results of this paper, a safety controller was implemented on a modified F1/10 race car (Figure 4a).

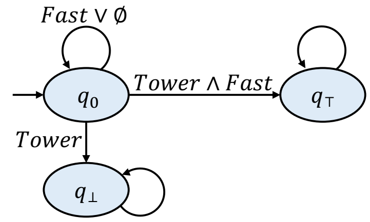


Fig. 3: Monitor $\mathcal{M}^\varphi$ for $\varphi = (\neg Tower)\, U\, (Tower \wedge Fast)$. Monitor states $q_\top$, $q_\perp$ and q_0 output *true*, *false* and *inconclusive*, respectively.

F1/10 is an open-source 1/10 scale autonomous vehicle testbed designed primarily for use by academic researchers [20].

When traveling along a straight line, the plant dynamics of the system conform to a non-deterministic discrete-time double integrator model. Here the system state at a time $k \in \mathbb{N}_{\geq 0}$, is described by the car's position along the road $x(k)$ and the car's forward velocity $v(k)$. Control inputs were suggested by a human operator, who chose the applied motor torque with a wireless Logitech Gamepad F710 controller (Figure 4a); this input is therefore proportional to the experienced acceleration. A non-deterministic factor was included in the system model in order to encapsulate the effects of drag on the vehicle.

Our mission objective is taken from the movie *Back to the Future*: when the car passes the clock tower, the car's velocity must be greater than 2 meters per second. We give the car, hereafter referred to as a DeLorean, an initial position $(x_0, v_0) = (0, 0)$, and arbitrarily place the clock tower a distance 2.54 meters from the origin.

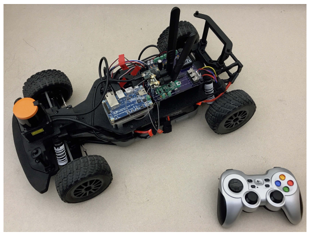
Without an enforcement mechanism, the human operator has the ability to suggest an input sequence which causes the vehicle to pass the clock tower with inadequate speed, thus violating the mission objective. We therefore design an assurance mechanism to act as a filter between the the human operator and the plant, assuring the system at runtime.

We convert our system specification to an LTL safety property as follows. Let $AP = \{Tower, Fast\}$ be the set of events, where $Tower$ indicates that the DeLorean has driven past the clock tower, and $Fast$ indicates that the velocity of the DeLorean is greater than 2 m/s. The trace of a system trajectory is therefore given by the labeling function $L : \mathbb{R}^2 \to 2^{AP}$

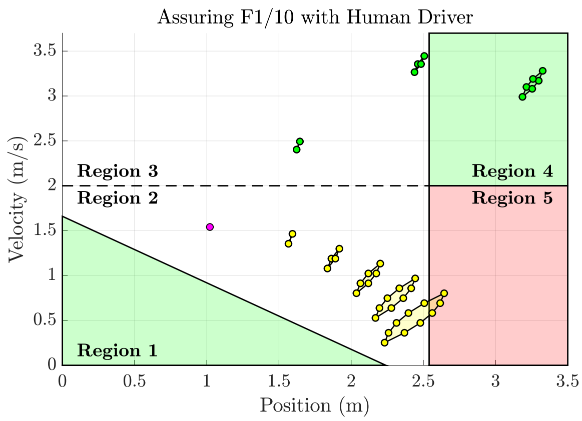
$$L(x, v) = \begin{cases} \varnothing & x < 2.54,\ v < 2 \\ Tower & x \geq 2.54,\ v < 2 \\ Fast & x < 2.54,\ v \geq 2 \\ Tower \wedge Fast & x \geq 2.54,\ v \geq 2. \end{cases}$$

In order to ensure that the DeLorean passes the clock tower with sufficient velocity, we enforce the LTL safety specification $\varphi = (\neg Tower)\, U\, (Tower \wedge Fast)$; see Figure 3. We take S^b to be a region of the state space from which the DeLorean can decelerate to zero velocity safely:

$$S^b = \{(x, v, q) \mid q = q_\top \ \textbf{ or } \ v \leq -0.69x + 1.66,\ q = q_?\}.$$



(a) Experimental test-bed. A human operator suggested performance control inputs to an modified F1/10 race car (left) using a Logitech Gamepad F710 controller (lower right).



(b) Safety controller implementation on F1/10 race car. The safe zone of the backup controller, S^b is shown in green (Regions 1 and 4). Regions 2, 3, 4 and 5 are labeled $\varnothing$, $Fast$, $Tower \wedge Fast$ and $Tower$, respectively. The current vehicle state is shown in pink. The non-deterministic trajectory resulting from a memorized recovery input sequence is shown in green, and the driver's suggested trajectory, which causes the system to violate the mission objective, is shown in yellow.

Fig. 4: Case study test-bed and trial data.

In this case, if the DeLorean is in a current state $(x, v, q) \in S^b$, then the backup controller will suggest that the DeLorean brake such that the vehicle decelerates to a stop. A safety controller architecture is created by integrating Algorithms 3 and 4 into an assurance mechanism.

We present the scenario where the vehicle driver, who initially suggested safe inputs, suggests an unsafe control policy (Figure 4b). Performance control inputs are passed to the system 250 milliseconds after the driver sent them via remote control; this lag-time allows the assurance mechanism to analyze control inputs as though they were suggested in a string. The driver first suggests an input sequence that guarantees that the DeLorean will satisfy the mission objective φ. This allows the DeLorean to leave S^b. At a future timestep, the driver suggests a control input sequence which allowed for the possibility that the DeLorean would violate φ. The assurance mechanism then applies the memorized recovery input sequence, and the DeLorean passes the clock tower with sufficient velocity.

VI. Conclusions

This paper introduces the safety controller as a runtime assurance mechanism for system objectives expressed as linear temporal logic properties. A case study is presented which details the construction and implementation of a safety controller on a non-deterministic cyber-physical system.

VII. Acknowledgements

The authors wish to thank Will Stuckey for his work with test-bed development.

References

[1] E. Bartocci, J. Deshmukh, A. Donzé, G. Fainekos, O. Maler, D. Ničković, and S. Sankaranarayanan, *Specification-Based Monitoring of Cyber-Physical Systems: A Survey on Theory, Tools and Applications*, pp. 135–175. Cham: Springer International Publishing, 2018.

[2] A. Bauer, M. Leucker, and C. Schallhart, "Runtime verification for LTL and TLTL," *ACM Trans. Softw. Eng. Methodol.*, vol. 20, pp. 14:1–14:64, Sept. 2011.

[3] D. Neider, M. Schwenger, P. Tabuada, A. Weinert, and M. Zimmermann, "Robust monitoring of linear temporal properties," *CoRR*, vol. abs/1807.08203, 2018.

[4] J. V. Deshmukh, A. Donzé, S. Ghosh, X. Jin, G. Juniwal, and S. A. Seshia, "Robust online monitoring of signal temporal logic," *Formal Methods in System Design*, vol. 51, pp. 5–30, Aug 2017.

[5] S. Mitra, Y. Wang, N. Lynch, and E. Feron, "Safety verification of model helicopter controller using hybrid input/output automata," in *Hybrid Systems: Computation and Control* (O. Maler and A. Pnueli, eds.), pp. 343–358, Springer Berlin Heidelberg, 2003.

[6] J. Ligatti, L. Bauer, and D. Walker, "Edit automata: enforcement mechanisms for run-time security policies," *International Journal of Information Security*, vol. 4, pp. 2–16, Feb 2005.

[7] N. Bielova and F. Massacci, "Do you really mean what you actually enforced?," in *Formal Aspects in Security and Trust* (P. Degano, J. Guttman, and F. Martinelli, eds.), (Berlin, Heidelberg), pp. 287–301, Springer Berlin Heidelberg, 2009.

[8] I. Mitchell and C. J. Tomlin, "Level set methods for computation in hybrid systems," in *Hybrid Systems: Computation and Control* (N. Lynch and B. H. Krogh, eds.), (Berlin, Heidelberg), pp. 310–323, Springer Berlin Heidelberg, 2000.

[9] S. Prajna and A. Jadbabaie, "Safety verification of hybrid systems using barrier certificates," in *Hybrid Systems: Computation and Control* (R. Alur and G. J. Pappas, eds.), (Berlin, Heidelberg), pp. 477–492, Springer Berlin Heidelberg, 2004.

[10] T. Wongpiromsarn, U. Topcu, and R. M. Murray, "Receding horizon temporal logic planning for dynamical systems," in *Proceedings of the 48h IEEE Conference on Decision and Control (CDC) held jointly with 2009 28th Chinese Control Conference*, pp. 5997–6004, Dec 2009.

[11] A. Jadbabaie, J. Yu, and J. Hauser, "Unconstrained receding-horizon control of nonlinear systems," *IEEE Transactions on Automatic Control*, vol. 46, pp. 776–783, May 2001.

[12] J. G. Rivera and A. A. Danylyszyn, "Formalizing the uni-processor simplex architecture," tech. rep., Carnegie Mellon University School of Computer Science, 1995.

[13] C. Belta, B. Yordanov, and E. A. Gol, *Formal Methods for Discrete-Time Dynamical Systems*, vol. 89. Springer, 2017.

[14] O. Kupferman and M. Y. Vardi, "Model checking of safety properties," *Formal Methods in System Design*, vol. 19, pp. 291–314, Nov 2001.

[15] C. Baier and J.-P. Katoen, *Principles of Model Checking*, vol. 26202649. MIT Press, 01 2008.

[16] J. Berstel, L. Boasson, O. Carton, and I. Fagnot, "Minimization of automata," *CoRR*, vol. abs/1010.5318, 2010.

[17] P. Tabuada, *Verification and Control of Hybrid Systems: A Symbolic Approach*. Springer Publishing Company, Incorporated, 1st ed., 2009.

[18] W. S. Levine, *The Control Systems Handbook: Control System Advanced Methods*. CRC press, 2010.

[19] M. Alshiekh, R. Bloem, R. Ehlers, B. Knighofer, S. Niekum, and U. Topcu, "Safe reinforcement learning via shielding," in *AAAI Conference on Artificial Intelligence*, 2018.

[20] M. O'Kelly, V. Sukhil, H. Abbas, J. Harkins, C. Kao, Y. V. Pant, R. Mangharam, D. Agarwal, M. Behl, P. Burgio, and M. Bertogna, "F1/10: An open-source autonomous cyber-physical platform," 2019.