Shield Synthesis for Real: Enforcing Safety in Cyber-Physical Systems

Meng Wu
Virginia Tech
Blacksburg, Virginia, USA

Jingbo Wang, Jyotirmoy Deshmukh, Chao Wang
University of Southern California
Los Angeles, California, USA

***Abstract*—Cyber-physical systems are often safety-critical in that violations of safety properties may lead to catastrophes. We propose a method to enforce the safety of systems with real-valued signals by synthesizing a runtime enforcer called the *shield*. Whenever the system violates a property, the shield, composed with the system, makes correction instantaneously to ensure that no erroneous output is generated by the combined system. While techniques for synthesizing Boolean shields are well understood, they do not handle real-valued signals ubiquitous in cyber-physical systems, meaning their corrections may be either *unrealizable* or *inefficient* to compute in the real domain. We solve the realizability and efficiency problems by analyzing the compatibility of predicates defined over real-valued signals, and using the analysis result to constrain a two-player safety game used to synthesize the shield. We demonstrate the effectiveness of this method on a variety of applications, including an automotive powertrain control system.**

***Index Terms*—program synthesis, controller synthesis, safety game, signal temporal logic, cyber physical system**

I. Introduction

A cyber-physical system often needs to continuously respond to external stimuli with actions under strict timing and safety requirements; violations of these requirements may lead to catastrophes. While formal verification is desirable, in practice, it can be difficult due to high system complexity, unavailability of source code, and limited capacity of existing verification tools. In addition, many systems have started incorporating machine learning components, which remain challenging to test or verify [13], [21].

Bloem et al. [7] recently introduced the concept of *shield*, denoted $\mathcal{S}$, to enforce a specification φ of a system $\mathcal{D}$ with certainty. The goal is to ensure that the combined $\mathcal{D} \circ \mathcal{S}$ never violates φ. If, for example, $\mathcal{D}$ malfunctions and produces an erroneous output O for input I, $\mathcal{S}$ will correct O into O' *instantaneously* to ensure $\varphi(I, O')$ holds even when $\varphi(I, O)$ fails. Here, instantaneously means correction is made in the same clock cycle. Furthermore, $\mathcal{S}$ depends solely on φ, which makes it well-suited for complex $\mathcal{D}$ but small φ, e.g., learning-based systems [2], [3], [42], [44].

While the functional specification of $\mathcal{D}$, denoted Ψ, may be large, typically, only a small subset $\varphi \subseteq \Psi$ is *safety-critical*. Since $\mathcal{S}$ depends on φ, as opposed to Ψ or $\mathcal{D}$, synthesizing $\mathcal{S}$ from φ is more practical than model checking [9], [33], which decides if $\mathcal{D}$ satisfies φ, or program synthesis [5], [6], [14], [32], which creates $\mathcal{D}$ from Ψ.

Although techniques for synthesizing Boolean shields are well understood [7], [22], [43], they do not work for systems where signals have real values and need to satisfy constraints such as $x + y \leq 1.53$. Naively treating the real-valued constraint as a predicate, or a Boolean variable P, may lead to *loss of information* at the synthesis time and *unrealizability* at run time. For example, while the Boolean combination $P \wedge \neg Q \wedge \neg R$ may be allowed, the corresponding real-valued constraint may not have solution, e.g., with $P : x + y \leq 1.53$, $Q : x < 1.0$ and $R : y < 1.0$.

Even the use of *abstraction refinement* to combine a Boolean shield with constraint solving does not work. For example, one may be tempted to block $P \wedge \neg Q \wedge \neg R$ and ask the shield to generate a new solution. However, since the shield must be reflexive, i.e., producing O' in the same clock cycle when the erroneous O occurs, it may be too slow at run time to recompute a solution. Even if it is fast enough, the new solution may still be unrealizable in the real domain. In general, it is difficult to bound *a priori* the number of iterations in such an *abstraction-refinement* loop to meet the strict timing requirement.

We propose a shield synthesis method to guarantee, with certainty, the realizability of real-valued signals. This is accomplished by treating Boolean and real-valued signals uniformly by adding a set of new constraints. These constraints take the form of two automata: a *relaxation automaton*, to capture the impossible combinations of predicates over signals in I and O, and a *feasibility automaton*, to capture the infeasible combinations of signals in O'. We use them to restrict the synthesis algorithm formulated as a two-player safety game, where the *antagonist* controls the erroneous O and the *protagonist* (shield) controls the corrected O': the game is won if the protagonist ensures that $\varphi(I, O')$ holds even if $\varphi(I, O)$ fails.

The overall flow is shown in Fig. 1, where the input consists of real-valued I_r and O_r signals and a safety property φ_r defined over these signals. Internally, the shield $\mathcal{S}$ has three subcomponents: a converter from real-valued I_r/O_r signals to Boolean I/O signals, a converter from Boolean O' signals to real-valued O'_r signals, and a Boolean shield $\mathcal{S}(I, O, O')$. Note that the system, denoted $\mathcal{D}(I_r, O_r)$, is not required to synthesize the shield: by treating $\mathcal{D}$ as a blackbox, we ensure that $\mathcal{D} \circ \mathcal{S} \models \varphi_r$ for any $\mathcal{D}$.

Our shield synthesis algorithm first computes a set $\mathcal{P}$ of predicates over real-valued signals from φ_r, I_r, O_r and O'_r. Next, it leverages $\mathcal{P}$ to construct the Boolean abstractions φ, I, O and O', as well as the relaxation automaton $\mathcal{R}(I, O)$ and the feasibility automaton $\mathcal{F}(O')$. Using these components, it constructs and solves a safety game where the antagonist is

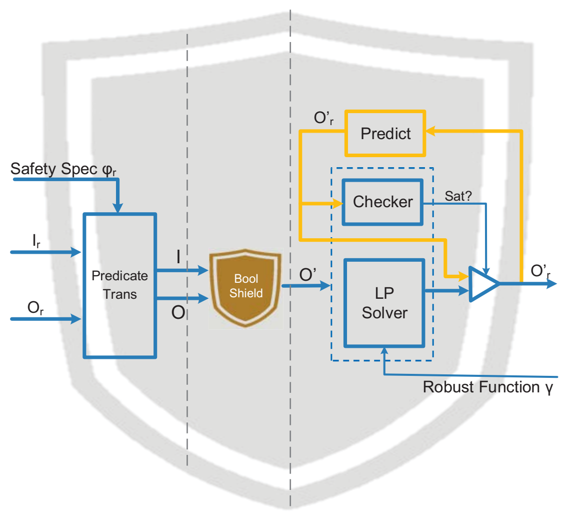

Fig. 1. Overview of the safety shield for real.

free to introduce errors to O and the protagonist must correct them in O'. The winning strategy computed for the protagonist is the Boolean shield $\mathcal{S}(I, O, O')$. At run time, real values are computed for signals in O'_r by solving a conjunction of constraints based on the Boolean values of signals in O.

To speed up the computation of real values at run time, we also propose a set of design-time optimizations, which leverage the information gathered from the shield to simplify the constraints to be solved at run time. When there are multiple real-valued solutions, the utility function γ shown in Fig. 1, which defines a *robustness* criterion, is used to pick the best one. We also propose a two-phase, *predict-and-validate* technique to speed up the computation of the real-valued solutions.

We have evaluated the method on a number of applications, including automotive powertrain control [19], autonomous driving [35], adaptive cruise control [29], multi-drone fleet control [30], generic control [20], blood glucose control [39], and water tank control [2]. Our results show that, in all cases, the shield can quickly produce real-valued correction signals at run time. Furthermore, the use of robustness constraints and two-phase computation can significantly improve the quality and efficiency of the real-valued solutions.

To sum up, we make the following contributions:

- We propose a method for synthesizing shields while guaranteeing the realizability of real-valued signals.
- We propose optimizations to speed up the computation and improve the quality of these correction signals.
- We demonstrate the effectiveness of the proposed techniques on a number of applications.

The remainder of this paper is organized as follows. First, we review the basics of shield synthesis in Section II. Then, we present the technical challenges of extending the Boolean shield to the real domain in Section III. Details of our method for addressing these challenges can be found in Sections IV and V. Next, we present our experimental results in Section VI. We review the related work in Section VII. Finally, we give our conclusions in Section VIII.

II. Preliminaries

We assume that the system, $\mathcal{D}$, is a blackbox with input I and output O. When $\mathcal{D}$ malfunctions, it produces some erroneous values in O. The shield, $\mathcal{S}$, takes both I and O as input and returns O' as output. Whenever $\mathcal{D} \models \varphi$, the shield returns $O' = O$; and when $\mathcal{D} \not\models \varphi$, the shield needs to compute correction O' for O. Following Bloem et al. [7], we treat the correction computation as a two-player safety game.

Safety Game The antagonist controls the alphabet Σ_{IO} and the protagonist controls the alphabet $\Sigma_{O'}$. The game is a tuple $\mathcal{G} = (G, g_0, \Sigma_{IO} \times \Sigma_{O'}, \delta_{\mathcal{G}}, F)$, where G is a set of states, g_0 is the initial state, $\Sigma_{IO} \times \Sigma_{O'}$ is the combined alphabet, $\delta_{\mathcal{G}} : G \times \Sigma_{IO} \times \Sigma_{O'} \to G$ is the transition function, and F is the set of unsafe states. In each state $g \in G$, the antagonist chooses a letter $\sigma_{IO} \in \Sigma_{IO}$ and then the protagonist chooses a letter $\sigma_{O'} \in \Sigma_{O'}$, thus leading to state $g' = \delta_{\mathcal{G}}(g, \sigma_{IO}, \sigma_{O'})$. The resulting state sequence $g_0 g_1 \ldots$ is called a play. A play is *winning* for the protagonist when it visits only the safe states.

The game may be solved using the classic algorithm of Mazala [28], which computes "attractors" for a subset of safe states $(G \setminus F)$ and unsafe states F. A winning *region* $\mathcal{W}$ is a subset of $(G \setminus F)$ states within which the protagonist has a strategy to win. A winning *strategy* is a function $\omega : G \times \Sigma_{IO} \to \Sigma_{O'}$ that ensures the protagonist always wins. The shield $\mathcal{S}$ is an implementation of the winning strategy.

Boolean Shield It is a tuple $\mathcal{S} = (S, s_0, \Sigma_{IO}, \Sigma_{O'}, \delta_{\mathcal{S}}, \lambda_{\mathcal{S}})$, where S is a set of states, s_0 is the initial state, $\delta_{\mathcal{S}} : S \times \Sigma_{IO} \to S$ is the transition function, and $\lambda_{\mathcal{S}}(S, \sigma_{IO}) = \sigma_{O'}$ is the output function. Here, $\lambda_{\mathcal{S}}$ implements the winning strategy ω in the game $\mathcal{G}$. Assume the system is $\mathcal{D} = (Q, q_0, \Sigma_I, \Sigma_O, \delta_{\mathcal{D}}, \lambda_{\mathcal{D}})$, where Q is the set of system states, q_0 is the initial state, $\delta_{\mathcal{D}} : Q \times \Sigma_I \to Q$ is the transition function, and $\lambda_{\mathcal{D}} : Q \times \Sigma_I \to \Sigma_O$ is the output function. The composition $\mathcal{D} \circ \mathcal{S}$ is $(QS, qs_0, \Sigma_I, \Sigma_{O'}, \delta_{\mathcal{D} \circ \mathcal{S}}, \lambda_{\mathcal{D} \circ \mathcal{S}})$, where $QS = Q \times S$, the initial state is $qs_0 = (q_0, s_0)$, the transition function is $\delta_{\mathcal{D} \circ \mathcal{S}} : QS \times \Sigma_I \to QS$, and the output function is $\lambda_{\mathcal{D} \circ \mathcal{S}}$.

Given a state $qs = (q, s)$, the next state $qs' = (q', s')$ is computed by $\delta_{\mathcal{D} \circ \mathcal{S}}(qs, \sigma_I)$ as follows: $q' = \delta_{\mathcal{D}}(q, \sigma_I)$ and $s' = \delta_{\mathcal{S}}(s, \sigma_I \circ \lambda_{\mathcal{D}}(\sigma_I))$, and $\sigma_{O'}$ is computed by $\lambda_{\mathcal{D} \circ \mathcal{S}}(qs, \sigma_I)$ as follows: $\lambda_{\mathcal{S}}(s, \sigma_I \circ \lambda_{\mathcal{D}}(\sigma_I))$.

If there are multiple ways of changing O to O' to satisfy $\varphi(I, O')$, the shield must choose the one with minimum difference between O and O'. The difference may be measured in Hamming Distance [7]: when $\mathcal{D} \models \varphi$, $HD(O, O') = 0$; and when $\mathcal{D} \not\models \varphi$, $HD(O, O')$ is minimized.

Example Consider the following two formulas in LTL [31]:

$$\mathsf{G}\Big(A \Rightarrow B_1\Big)$$
$$\mathsf{G}\Big(A \wedge \mathsf{X}(\neg A) \Rightarrow B_2 \mathsf{U} A\Big)$$

where G means *Globally*, X means *Next*, U means *Until*, Boolean variable A is an input signal, while B_1 and B_2 are output signals of $\mathcal{D}$. Fig. 2 shows the corresponding automaton

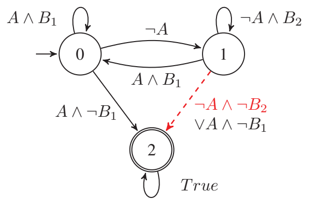

Fig. 2. Boolean safety specification.

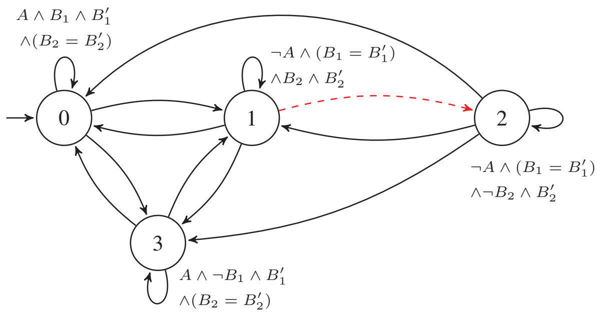
$A \wedge B_1 \wedge B_1'$
$\wedge (B_2 = B_2')$
$\neg A \wedge (B_1 = B_1')$
$\wedge B_2 \wedge B_2'$
0
1
2
$\neg A \wedge (B_1 = B_1')$
$\wedge \neg B_2 \wedge B_2'$
3
$A \wedge \neg B_1 \wedge B_1'$
$\wedge (B_2 = B_2')$

Fig. 3. Boolean shield for Properties in Fig. 2.

representation, where 0 is the initial state and 2 is an unsafe state. Note that the transition labels are on the edges.

Fig. 3 shows the shield generated by existing techniques [7], [22], [43], which takes signals A, B_1 and B_2 as input, and return the modified signals B_1' and B_2' as output. Here, labels are on nodes instead of edges: they are conditions under which transitions go to destination nodes. Furthermore, $B_1 = B_1'$ means the two signals have the same value.

The shield ensures that A, the input signal of $\mathcal{D}$, and B_1', B_2', the modified output signals of $\mathcal{D}$, always satisfy the specification in Fig. 2. At the same time, the deviation between B_1, B_2 and B_1', B_2' is minimized.

The red dashed edge in Fig. 3 illustrates a scenario where $\mathcal{D}$ violates the specification by setting B_2 to *false* while moving from a state where A is true to a state where A is false, as represented by the red dashed edge in Fig. 2. The shield, upon detecting this violation, responds instantaneously by setting B_2' to *true*. It allows the specification to be satisfied by the modified output (B_2'). As for B_1', whose value does not matter, the shield maintains $B_1' = B_1$ to minimize the deviation.

III. Technical Challenges

Using a Boolean shield to generate real-valued correction signals has two problems: realizability of the Boolean predicates, and quality of the real-valued signals.

A. Realizability of the Boolean Predicates

The Boolean specification in Section II are abstractions of the real-valued LTL properties below, which in turn are abstractions of properties of an automotive powertrain control system [19] expressed in Signal Temporal Logic (STL [27]).

$$\mathsf{G}\big(l = \mathsf{power} \Rightarrow |\mu| < 0.2\big)$$

$$\mathsf{G}\Big(l = \mathsf{power} \wedge \mathsf{X}(l = \mathsf{normal}) \Rightarrow \big(|\mu| < 0.02\big)\,\mathsf{U}\big(l = \mathsf{power}\big)\Big)$$

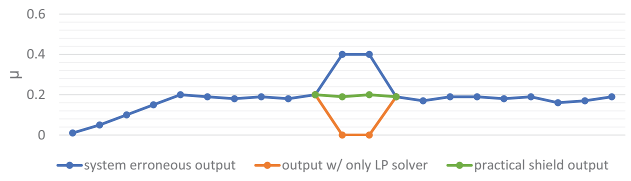

Fig. 4. Importance of the smoothness in real-valued correction signals.

The input signal l denotes the system mode, which may be normal or power. The output signal μ is the normalized error of the air-fuel (A/F) ratio inside an internal combustion engine. Let λ be the A/F ratio and λ_{ref} be a reference value, then $\mu = (\lambda - \lambda_{ref})/\lambda_{ref}$. Since μ affects other parts of the systems, it must be kept in certain regions depending on the system mode.

The first property says that $|\mu|$ should stay below 0.2 in the power mode. The second property says that, after the system changes from the power mode to the normal mode, $|\mu|$ should stay below 0.02. In the Boolean versions, A denotes whether the system is in the power mode, while B_1 and B_2 denote $|\mu| < 0.2$ and $|\mu| < 0.02$, respectively. The combination $\neg B_1 \wedge B_2$ is unrealizable, because $|\mu|$ cannot be both greater than 0.2 and less than 0.02.

However, the shield synthesized by existing methods is not aware of this problem, and thus may produce combinations of Boolean values that are not realizable in the real domain. As shown by the red edge in Fig. 3: if the shield's input is $\neg A \wedge \neg B_1 \wedge \neg B_2$, the shield's output will be $\neg B_1' \wedge B_2'$, despite that $|\mu'| \geq 0.2 \wedge |\mu'| < 0.02$ is unsatisfiable.

We solve this problem by checking the compatibility of the predicates at the synthesis time, to guarantee their realizability at run time. Details will be presented in Section IV.

B. Quality of the Real-valued Output

Even if the Boolean values are realizable, the real-valued solution computed by a generic solver may not be of high quality. Assume that all predicates are linear arithmetic constraints, the output of a Boolean shield would be a conjunction of constraints. As shown in Fig. 1, the back-end may convert O', the Boolean output, to O_r', the real-valued output, by solving a linear programming (LP) problem.

Consider $\mathsf{G}(A \Rightarrow B)$, which abstracts $\mathsf{G}\big(l = \mathsf{power} \Rightarrow |\mu| < 0.2\big)$. Suppose the original system's output violates the property $|\mu| < 0.2$ as shown by the blue line in Fig. 4, where the two erroneous values are in the middle. The correction computed by an LP solver may be any of the infinitely many values in the interval (-0.2, +0.2), including -0.19 and 0. However, neither of these two values may be acceptable in a real system, which expects the signal to be *stable*, not *arbitrary*.

Ideally, we want to generate real-valued signals that are smooth, and consistent with physical laws of the environment, e.g., the green line in Fig. 4. Toward this end, we leverage a utility function, γ, to impose *robustness* in addition to *correctness* constraints. With both types of constraints, the LP solver can generate values of high quality.

Algorithm 1 Synthesizing a realizable Boolean shield $\mathcal{S}_{bool}$ from φ_r.

```
1: Let $\mathcal{P}$ be the set of predicates over real-valued variables in $\varphi_r$;
2: Let $\varphi$, $I$, $O$, $O'$ be Boolean abstractions of $\varphi_r$, $I_r$, $O_r$, $O'_r$ via $\mathcal{P}$;
3: function SYNTHESIZEBOOL ( $\mathcal{P}$, $I$, $O$, $O'$ )
4:     $\mathcal{Q}(I, O') \leftarrow$ GENCORRECTNESSMONITOR($\varphi$)
5:     $\mathcal{E}(I, O, O') \leftarrow$ GENERRORAVOIDINGMONITOR($\varphi$)
6:     $\mathcal{G} \leftarrow \mathcal{Q} \circ \mathcal{E}$
7:     $\mathcal{W} \leftarrow$ COMPUTEWINNINGSTRATEGY($\mathcal{G}$)
8:     $\mathcal{R}(I, O) \leftarrow$ GENRELAXATIONAUTOMATON($P, I, O, \mathcal{W}$)
9:     $\mathcal{F}(O') \leftarrow$ GENFEASIBILITYAUTOMATON($\mathcal{R}$)
10:    $\mathcal{G}_r \leftarrow \mathcal{W} \circ \mathcal{R} \circ \mathcal{F}$
11:    $\omega_r \leftarrow$ COMPUTEWINNINGSTRATEGY($\mathcal{G}_r$)
12:    $\mathcal{S}_{bool}(I, O, O') \leftarrow$ IMPLEMENTSHIELD($\omega_r$)
13:    return $\mathcal{S}_{bool}$
14: end function
```

We also propose a technique to speed up the computation of real values. The intuition is that system dynamics may be approximated using (linear) regression, which predicts the current value of a signal based on its values in the recent past. Thus, we develop a fast runtime prediction unit to guess the value, followed by a fast validation unit to check its validity. If the predicted value is valid, it will serve as the shield's output. Otherwise, we invoke the LP solver. Details will be presented in Section V.

IV. Synthesizing the Boolean Shield

In this section, we present our method for ensuring the realizability of the shield's output signals. The idea is to check the compatibility of predicates inside the game-based algorithm for synthesizing the Boolean shield. To improve efficiency, we check predicate combinations only when they are involved in compute the winning strategy.

Algorithm 1 shows the procedure, where blue highlighted lines address the *realizability* issue while the remainder follows the classic algorithm in the prior work [7], [22], [43]. First, it creates $\mathcal{P}$, the set of predicates from the real-valued specification φ_r. Then, it uses $\mathcal{P}$ to compute a Boolean abstraction of φ_r, denoted φ. Next, it uses φ to formulate a two-player safety game $\mathcal{G}$ where the antagonist controls I and O, the protagonist controls O', and $\mathcal{W}$ is the winning region where the protagonist may win the game.

Since the construction of the safety game $\mathcal{G}$ is part of the prior work, we refer to Bloem et al. [7] and Wu et al. [43] for details. Here, it suffices to say that $\mathcal{G}$ is a synchronous composition of $\mathcal{E}$, an error-avoiding monitor that outlines all possible ways in which the antagonist may introduce errors in O and the protagonist may introduce corrections in O', and $\mathcal{Q}$, a correctness monitor that ensures $\varphi(I, O')$ always holds.

Since a winning strategy in $\mathcal{W}$ may not be realizable in the real domain, our next step is to compute a strategy ω_r based on $\mathcal{W}$ while ensuring correction signals produced by ω_r are always realizable. Toward this end, we introduce two additional automata: the *feasibility* automaton $\mathcal{F}(O')$ and the *relaxation* automaton $\mathcal{R}(I, O)$. Specifically, $\mathcal{F}$ is used to identify and remove the infeasible edges in ω, i.e., corrections in O' with no real-valued solutions. $\mathcal{R}$ is used to identify and remove the unrealistic errors in I and O, i.e., errors that are impossible and thus will not occur in the first place.

In other words, $\mathcal{F}$ restricts the search to realizable solutions, and $\mathcal{R}$ allows us to have more freedom while computing the winning strategy. Thus, the new game $\mathcal{G}_r$ is a composition of $\mathcal{W}$, $\mathcal{R}$ and $\mathcal{F}$. Based on the winning strategy ω_r computed from $\mathcal{G}_r$, we can construct a shield $\mathcal{S}_{bool}$ that is guaranteed to be realizable at run time.

In the remainder of this section, we illustrate the details while focusing on the highlighted lines in Algorithm 1.

A. Computing the Predicates

$\mathcal{P}$ is the set of predicates over real-valued signals used in φ_r, where φ_r is expressed in Signal Temporal Logic (STL). In addition to the LTL operators, STL also has dense time intervals associated with temporal operators and constraints over real-valued variables.

Consider the STL formulas below, which come from the powertrain control system [19] without modification.

$$\mathsf{G}_{[\tau_s,T]}(l = \mathsf{power} \Rightarrow |\mu| < 0.2)$$
$$\mathsf{G}_{[\tau_s,T]}\Big(l = \mathsf{power} \wedge \mathsf{X}(l = \mathsf{normal}) \Rightarrow \mathsf{G}_{[\eta, \frac{\varsigma}{2}]}(|\mu| < 0.02)\Big)$$

Here, $\mathsf{G}_{[\tau_1,\tau_2]}$ is the temporal operator augmented with time interval $[\tau_1, \tau_2]$, l is the system mode, and μ is the normalized error of the air-fuel ratio. The first property says that $|\mu|$ should stay below 0.2 immediately after the system switch to the power mode, i.e., between time τ_s and time T. The second property says that, when it switches from the power mode to the normal mode, $|\mu|$ should settle down to below 0.02 after time η and before time $\frac{\varsigma}{2}$.

To compute $\mathcal{P}$, first, we convert each time interval to a conjunction of linear constraints, e.g., by using a time variable t to represent the bounds in intervals $[\tau_s, T]$ and $[\eta, \frac{\varsigma}{2}]$.

T_1:	$(t \geq \tau_s)$	T_2:	$(t \leq T)$
T_3:	$(t \geq \eta)$	T_4:	$(t \leq \frac{\varsigma}{2})$

Next, we convert the constraints over real-valued variables to predicates. From the running example, we will produce the following predicates:

L_1:	$(l = \mathsf{power})$	L_2:	$(l = \mathsf{normal})$
M_1:	$(\lvert\mu\rvert < 0.2)$	M_2:	$(\lvert\mu\rvert < 0.02)$

B. Computing the Boolean Abstractions

After the set $\mathcal{P}$ of predicates is computed, we use it to compute the Boolean abstractions of φ_r, I_r, O_r and O'_r. This step is straightforward. To compute φ from φ_r, we traverse the abstract syntax tree (AST) of φ_r and, for each AST node n that corresponds to a real-valued predicate $P \in \mathcal{P}$, we replace P with a new Boolean variable v_P.

To compute I from I_r, we traverse the predicates in $\mathcal{P}$ and, for each predicate $Q \in \mathcal{P}$ defined over some real-valued signals in I_r, we add a new Boolean variable v_Q to I. Similarly, O and O' are also computed from O_r and O'_r by creating new Boolean variables.

C. Computing the Relaxation Automaton

The relaxation automaton $\mathcal{R}$ aims to identify impossible combinations of I and O values, and since they will never occur in the shield's input, there is no need to make corrections in the shield's output. There may be two reasons why a value combination is impossible:

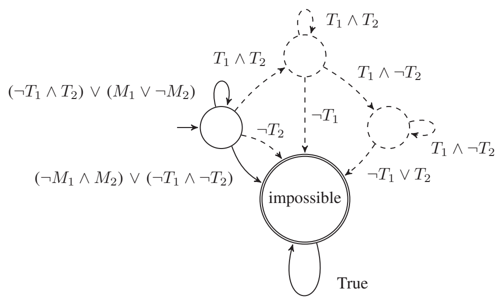

Fig. 5. Relaxation automaton $\mathcal{R}(I, O)$: *impossible* means the system $\mathcal{D}$ will not allow the state to be reached, and the shield $\mathcal{S}$ can treat it as *don't care*.

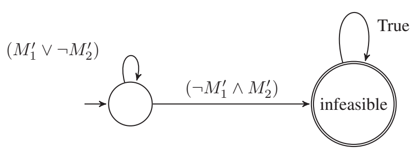

Fig. 6. Feasibility automaton $\mathcal{F}(O')$: *infeasible* means the state is unrealizable, and the shield $\mathcal{S}$ must avoid the related edges while generating solutions.

1) The values of real-valued predicates are incompatible, e.g., as in $|\mu| < 0.02$ and $|\mu| > 0.2$.
2) The values are not consistent with physical laws of the environment, e.g., time never travels backward. For example, with respect to the time interval $[\tau_s, T]$, the transition from $T_1 \wedge T_2$ to $\neg T_1 \wedge T_2$ is impossible.

In addition, our method allows users to provide more constraints to characterize physical laws of the environment or their understanding of the behaviors of the system $\mathcal{D}$.

States in the relaxation automaton $\mathcal{R}$ are divided into two types: *normal* states and *impossible* states. Here, *normal* means the I/O behavior of the system $\mathcal{D}$ may occur, whereas *impossible* means it will never occur. Since impossible I/O behavior will never occur in the shield's input, the shield may treat it as *don't-care* and thus have more freedom to compute the winning strategy.

Example Fig. 5 shows the relaxation automaton for our running example. Here, the dashed edges come from the physical laws (time never travels backward), while the solid edges comes from the compatibility of real-valued predicates defined over l and μ. In particular, the combination $\neg M_1 \wedge M2$ is identified as impossible, because $|\mu|$ cannot be greater than 0.2 and less than 0.02 at the same time.

To check the compatibility of the predicate values, conceptually, one can iterate through all possible value combinations for the predicates in $\mathcal{P}$, and check each combination with an LP solver. If the combination is *unsatisfiable (UNSAT)* according to the LP solver, we say it is impossible. However, in our actual implementation, the compatibility checking is performed significantly more efficiently, due to the use of variable partitioning and UNSAT cores. First, $\mathcal{P}$ may be divided into subgroups, such that predicates from different subgroups do not interfere with each other. Therefore, value combinations may be computed via Cartesian products. Second, when a value combination is proved to be unsatisfiable, we compute its UNSAT core, i.e., a minimal subset that itself is UNSAT. By leveraging these UNSAT cores, we can significantly speed up the checking of value combinations.

D. Computing the Feasibility Automaton

The feasibility automaton $\mathcal{F}$ aims to capture the combinations of O' values that are unrealizable in the real domain. Similar to $\mathcal{R}$, states in $\mathcal{F}$ are divided into two types: *normal* and *infeasible*. Here, *normal* means the value combinations are realizable in the real domain, whereas *infeasible* means the value combinations may be unrealizable.

Fig. 6 shows an example feasibility automaton for the running example: all predicates are the primed versions because they are defined over O' signals, which are part of the modified output of the shield. Upon $\neg M_1' \wedge M_2'$, the automaton goes into the infeasible state, because $\neg(|\mu'| < 0.2) \wedge (|\mu'| < 0.02)$ has no real-valued solution.

While this is rare, a value combination may depend on real-valued signals not only in the shield's output (O_r') but also in the input (I_r). Let such a value combination be denoted by $\phi(I_r, O_r')$. *Whether ϕ is guaranteed to be realizable* can be decided using an SMT solver, by checking the validity of the formula $\forall I_r.\exists O_r'.\phi(I_r, O_r')$.

Subsequently, during our computation of the winning strategy ω_r, we need to avoid such unrealizable combinations.

E. Solving the Constrained Game

The new safety game $\mathcal{G}_r$ is defined as the composition of $\mathcal{W}$, the winning region of the Boolean game $\mathcal{G}$, the relaxation automaton $\mathcal{R}$, and the feasibility automaton $\mathcal{F}$. We tweak the winning region automaton $\mathcal{W}$ by adding an *unsafe* state for all edges going out of $\mathcal{W}$. Here, composition means the standard synchronous product, where a state transition exists only if it is allowed by all three components ($\mathcal{W}$, $\mathcal{R}$ and $\mathcal{F}$). Furthermore, safe states of $\mathcal{G}_r$ are either (1) states that are both *safe* in $\mathcal{W}$ and *feasible* in $\mathcal{F}$, or (2) states that are *impossible* in $\mathcal{R}$.

More formally, assume that $F^{\mathcal{W}}$ is the set of unsafe states related to the winning region $\mathcal{W}$, $F^{\mathcal{F}}$ is the set of infeasible states of the feasibility automaton $\mathcal{F}$, and $F^{\mathcal{R}}$ is the set of the impossible states of the relaxation automaton $\mathcal{R}$. The set of safe states in the new game $\mathcal{G}_r$ is defined as $(\neg F^{\mathcal{W}} \wedge \neg F^{\mathcal{F}}) \vee F^{\mathcal{R}}$.

Finally, we solve $\mathcal{G}_r$ using standard algorithms for safety games, e.g., Mazala [28], which are also used in the prior work [7], [22], [43]. The result is a winning strategy ω_r, which in turn may be implemented as a reactive component $\mathcal{S}_{bool}$. Note that $\mathcal{S}_{bool}$ is a Mealy machine that takes I and O signals as input and returns the modified O' signals as output. Furthermore, due to the use of $\mathcal{R}$ and $\mathcal{F}$, the output of $\mathcal{S}_{bool}$ is guaranteed to be realizable at run time.

V. Generating the Real-valued Signals

In this section, we present our method for computing the real-valued signals (O_r') at run time, based on the Boolean shield's output (O').

Algorithm 2 shows the details of our method, which needs I_r, O_r, O_r', the set $\mathcal{P}$ of predicates, $\mathcal{S}_{bool}$, and a utility function γ, which is used to evaluate the quality of the real-valued

Algorithm 2 Computing real-valued correction signals at run time.

```
1: function COMPUTEREALVALUES( $I_r, O_r, O'_r, \mathcal{P}, \mathcal{S}_{bool}, \gamma$)
2:     $I, O \leftarrow$ GENBOOLEANABSTRACTION($I_r, O_r, \mathcal{P}$)
3:     $O' \leftarrow$ GENBOOLEANSHIELDOUTPUT($\mathcal{S}_{bool}, I, O$)
4:     if $O' = O$ then
5:         $O'_r = O_r$
6:     else
7:         $O'_r \leftarrow$ PREDICTION($Hist$)
8:         if $\neg$ SATISFIABLE($\mathcal{P}, O', O'_r$) then
9:             $model \leftarrow$LPSOLVE($\mathcal{P}, \gamma, O'$)
10:            $O'_r \leftarrow model$
11:        end if
12:    end if
13:    $Hist \leftarrow Hist \cup \{O'_r\}$
14: end function
```

solution. First, real values in I_r and O_r are transformed to Boolean values in I and O. Then, they are used by $\mathcal{S}_{bool}$ to compute new values in O'. When O' and O have the same Boolean value, meaning the shield does not make any correction, O'_r and O_r will also have the same real value; in this case, no computation is needed (Line 5). However, when O' and O have different values, we need to recompute the real values in O'_r (Lines 7-11).

A. Robustness Optimization

Since the output of the Boolean shield is an assignment of the Boolean predicates in O', and each predicate corresponds to a linear constraint of the form $\Sigma_{i=1}^{k} a_i x_i \leq 0$, conceptually, the real values in O'_r can be computed by solving the linear programming (LP) problem.

However, naively invoking the LP solver does not always produce a high-quality solution. Instead, we develop the following optimization to improve the quality of the solution. Specifically, we restrict the LP problem using a robustness constraint derived from the utility function γ. While there may be various ways of defining robustness, especially in the context of STL [12], [15], a straightforward way that works in practice is to ensure the signal is *smooth* (see Fig. 4).

That is, we restrict the LP problem using the objective function

$$\min\left(|val^i - \frac{\sum_{k=1}^{N} val^{i-k}}{N}|\right)$$

where val^i denotes the current value (at the i-th time step) and val^{i-k}, where $k = 1, 2, \ldots$, denotes the value in the recent past. The above function aims to minimize the distance between val^i and the (moving) average of the previous N values, stored in $Hist$ (Line 13).

B. Value Prediction and Validation

While the robustness constraint improves the quality of the real-valued solution, it also increases the computational cost of LP solving. To reduce the computational cost, we develop a two-phase optimization for computing the solution.

First, we predict the value of a signal using standard regression algorithms based on the historical values of the signal in the immediate past (Line 7 in Algorithm 2). Here, the procedure PREDICTION leverages historical values stored in $Hist$. Since the signal is expected to be *smooth*, standard linear or non-linear regression can be very accurate in practice.

Next, we validate the predicted value (Line 8). This is accomplished by plugging the predicted value for O'_r into the combination of Boolean predicates defined by $\mathcal{P}$ and the values of signals in O'. If it is valid, the value is accepted as the final output, and invocation of the LP solver is avoided. Note that the time taken to perform prediction and validation is significantly smaller than that of the LP solving.

Only when the predicted value is not valid, we invoke the LP solver (Line 9). Even in this case, the response time is fast because we can use the same LP solver for validation and LP solving. Due to incremental computation inside the solver, the solution used for validation, which is often close to the final solution, can help speed up LP solving.

VI. EXPERIMENTS

We have implemented our method as a tool that takes the automaton representation of a safety specification as input and returns a real-valued shield as output. Internally, we solve the safety game using Mazala's algorithm [28] implemented symbolically using CUDD [1], and use the LP solver integrated in Z3 [11] for prediction, validation and constraint solving. For evaluation purposes, the shield is implemented as a C program and is executed at every time step. Each execution has two phases: (1) generating Boolean values for signals in O', and (2) generating real values for signals in O'_r.

Benchmarks We evaluated our tool on seven sets of benchmarks, including automotive powertrain control [19], autonomous driving [35], adaptive cruise control [29], multi-drone fleet control [30], generic control [20], blood glucose control [39], and water tank control [2]. In all benchmarks, the original specification was given in STL, which has both timing and real-valued constraints.

Table I shows the benchmark statistics, including the application name, the property, a short description, and the corresponding STL formula. For brevity, we omit the automaton representations. We conducted experiments on a computer with Intel i5 3.1GHz CPU, 8GB RAM, and the Ubuntu 14.04 operating system. Our experiments were designed to answer the following questions: (1) Is our tool efficient in synthesizing the real-valued shield? (2) Is the shield effective in preventing safety violations? (3) Are the real-valued signals produced by the shield of high quality?

Experimental Results Table II shows the results of our shield synthesis procedure. Columns 1-3 show the property name, the number of states of the specification, and the number of real-valued signals in I_r and O_r, respectively. Column 4 shows the number of predicates defined over signals in I_r and O_r. Based on these predicates, Boolean signals in I and O are created; Column 5 shows the number of these signals. Column 6 shows the number of conflicting constraints captured by the relaxation and feasibility automata, respectively. Column 7 shows the synthesis time. Columns 8-9 show the number of states of the Boolean shield, and the number of real-valued constraints to be solved at run time.

Table III shows the performance of the shields. For each shield, we generated input signals (for I_r and O_r) based on the system description: some input signals satisfy the specification while others do not. By measuring the response time of the

TABLE I
STATISTICS OF THE BENCHMARK APPLICATIONS.

Application	Property	STL Formula and Description
Powertrain	R26	In normal mode, permitted overshoot/undershoot is always less than 0.05 $\mathsf{G}_{[\tau_s,T]}(l{=}\mathsf{normal} \Rightarrow \lvert\mu\rvert < 0.05)$
	R27	In normal mode, overshoot/undershoot less than 0.02 within the settling time $\mathsf{G}_{[\tau_s,T]}\Big(\mathsf{rise}(a)\lvert\mathsf{fall}(a) \Rightarrow \mathsf{G}_{[\eta,\frac{\varsigma}{2}]}(\lvert\mu\rvert < 0.02)\Big)$
	R32	From power to normal, overshoot/undershoot less than 0.02 within settling time $\mathsf{G}_{[\tau_s,T]}\Big(l{=}\mathsf{power} \wedge \mathsf{X}(l{=}\mathsf{normal}) \Rightarrow \mathsf{G}_{[\eta,\frac{\varsigma}{2}]}(\lvert\mu\rvert < 0.02)\Big)$
	R33	In power mode, permitted overshoot or undershoot should be less than 0.2 $\mathsf{G}_{[\tau_s,T]}(l{=}\mathsf{power} \Rightarrow \lvert\mu\rvert < 0.2)$
	R34	Upon startup/sensor failure, overshoot/undershoot <0.1 within the settling time $\mathsf{G}_{[\tau_s,T]} l{=}\mathsf{startup}\lvert\mathsf{sensor\_fail} \wedge \mathsf{rise}(a)\lvert\mathsf{fall}(a) \Rightarrow \mathsf{G}_{[\eta,\frac{\varsigma}{2}]}(\lvert\mu\rvert < 0.1)\Big)$
Autonomous Driving	D1	Vehicle should keep a steady speed V_s when there is no collision risk $\mathsf{G}\big(\lvert y_k^{ego} - x_k^{adv}\rvert >= 4\big) \Rightarrow \mathsf{G}\big(\lvert v_k^{ego} - V_s\rvert < \varepsilon\big)$
	D2	Vehicle should come to stop for at least 2 second when there is collision risk $\mathsf{G}\big(\lvert y_k^{ego} - x_k^{adv}\rvert < 4\big) \Rightarrow \mathsf{G}_{[0,2]}\big(\lvert v_k^{ego}\rvert < 0.1\big)$
Cruise Control	A1	Keep a safe distance with lead vehicle: $\mathsf{G}\big(\mathsf{pos\_lead}[t] - \mathsf{pos\_ego}[t] > D_s\big)$
	A2	Achieve cruise velocity if there is a comfortable distance $\big(\mathsf{pos\_lead}[t] - \mathsf{pos\_ego}[t] > D_c\big)\mathsf{U}_{[0,10]}\big(\lvert\mathsf{v\_ego}[t] - \mathsf{v\_cruise}[t]\rvert < \varepsilon\big)$
	A3	Vehicle should never travel backward: $\mathsf{G}\big(\mathsf{v\_ego}[t] >= 0\big)$
	A4	Vehicle doesn't halt unless lead vehicle halts: $\mathsf{G}\big(\mathsf{v\_lead}[t] > 0\big) \Rightarrow \mathsf{G}\big(\mathsf{v\_ego}[t] > 0\big)$
Quadrotor Control	Q1	Drone flies to goal point if no obstacles are on the way: $\mathsf{G}\big(Obs(\mathbf{pos}^{quad}, \mathbf{pos}^{obs}) \Rightarrow \omega_g > 0)\big)$
	Q2	Avoiding obstacles: $\mathsf{G}\neg Obs(\mathbf{pos}^{quad}, \mathbf{pos}^{obs}) \Rightarrow \Big(\omega_{\bar{g}} > 0 \wedge \mathsf{G}\big(Dis(\mathbf{pos}^{quad}, \mathbf{pos}^{obs}) < \varepsilon \Rightarrow \omega_g = 0\big)\Big)$
General Control	C1	After settling, output error should be less than set value ε_b: $\mathsf{G}\Big(\mathsf{x}[t] \Rightarrow \mathsf{G}_{[10,\infty]}\big(\lvert\frac{\mathsf{y}[t]-\mathsf{y}^{ref}}{\mathsf{y}^{ref}}\rvert < \varepsilon_b\big)\Big)$
	C2	Output error should be $[\varepsilon^{\perp}, \varepsilon^{\top}]$ in settling time: $\mathsf{G}\Big(\mathsf{x}[t] \Rightarrow \mathsf{G}_{[0,20]}\big(\varepsilon^{\perp} < \frac{\mathsf{y}[t]-\mathsf{y}^{ref}}{\mathsf{y}^{ref}} < \varepsilon^{\top}\big)\Big)$
	C3	Output should achieve reference value within *rise_time*: $\mathsf{G}\Big(\mathsf{x}[t] \Rightarrow \mathsf{F}_{[0,\mathsf{rise\_time}]}\big(\lvert\frac{\mathsf{y}[t]-\mathsf{y}^{ref}}{\mathsf{y}^{ref}}\rvert < \varepsilon_r\big)\Big)$
Glucose Control	B1	Having meal within t_1 minutes after taking the bolus is safe. A bolus must be taken after t_2 minutes of having meal, if it is not yet taken: $\mathsf{G}\Big(\mathsf{F}_{[0,t_1+t_2]}(B > \mathsf{c}_2) \vee \mathsf{G}_{[t_1,t_1+t_2]}\big(M > \mathsf{c}_1 \Rightarrow \mathsf{F}_{[0,t_2]}(B > \mathsf{c}_2)\big)\Big)$
Water Tank Control	W1	Turn on inflow and turn off outflow switch when water level is low ($l < 4$) $\mathsf{G}\big(l < 4 \Rightarrow \mathsf{G}_{[0,3]}(\mathsf{flow}_{out} = 0 \wedge 1 < \mathsf{flow}_{in} < 2)\big)$
	W2	Turn on outflow and turn off inflow switch when water level is high ($l > 93$) $\mathsf{G}\big(l > 93 \Rightarrow \mathsf{G}_{[0,3]}(\mathsf{flow}_{in} = 0 \wedge 0 < \mathsf{flow}_{out} < 1)\big)$

TABLE II
RESULTS OF OUR NEW SHIELD SYNTHESIS PROCEDURE.

Name	Specification		Synthesis Tool				Shield $\mathcal{S}$	
	states	$\lvert I_r\rvert/\lvert O_r\rvert$	$\lvert\mathcal{P}_I\rvert/\lvert\mathcal{P}_O\rvert$	$\lvert I\rvert/\lvert O\rvert$	$\lvert\mathcal{R}\rvert/\lvert\mathcal{F}\rvert$	time(s)	states	constrs
R26+R27	8	1/1	2/2	5/2	2/1	0.16	25	2+2
R32+R33	9	1/1	2/2	5/2	2/1	0.15	28	2+2
R26+R27+R32 +R33+R34	23	1/1	2/4	5/4	12/11	1.15	158	4+2
D1	6	3/1	5/3	6/3	53/5	0.15	19	3+2
D2	5	3/1	2/3	3/3	5/5	0.21	30	3+2
D1+D2	14	3/1	5/3	6/3	53/5	0.8	164	3+2
A1+A3+A4	3	3/1	2/2	2/3	1/1	0.08	8	2+0
A2+A3+A4	4	4/1	3/3	3/3	4/4	0.1	15	3+0
A1+A2+A3+A4	7	4/1	4/3	4/4	8/4	0.55	48	3+0
Q1+Q2	5	1/2	1/2	2/2	0/0	0.08	7	2+0
C1+C2+C3	19	2/1	3/4	3/4	13/11	0.52	118	4+2
B1	5	3/1	5/1	5/1	14/0	0.1	6	1+0
W1+W2	6	1/2	2/2	2/2	1/0	0.1	10	2+2

shield under these input signals, and the quality of corrections made by the shield, we hope to evaluate its effectiveness.

In this table, Column 1 shows the property name. Column 2 shows the size of the C program that implements the shield. Column 3 shows the response time of the Boolean shield on input signals that do not violate the specification. Columns 4-5 show the response time on input signals that violate the specification. Among these columns, *prediction* means the real-valued solution was successfully computed by a linear regression, whereas *constraint solving* means prediction failed and the solution was computed by the LP solver.

Overall, the time to compute real-valued correction signals is within 0.5 ms when $\mathcal{D} \not\models \varphi$, and less than 1 us when $\mathcal{D} \models \varphi$. In the latter case, the shield does not need to make correction at all. In both cases, the response time is always bounded and fast enough for the target applications.

TABLE III
RESULTS OF EVALUATING RUNTIME PERFORMANCE OF THE SHIELD.

Name	Implementation	Shield Response Time		
	(LoC)	Boolean step (us)	prediction step (us)	constraint solving (us)
R26+R27	745	0.3	293.3	336.8
R32+R33	748	0.41	256.5	333.9
R26+R27+R32 +R33+R34	1446	0.8	245.0	279.8
D1	781	0.45	177.2	164.7
D2	853	0.5	313.5	329.0
D1+D2	2242	0.8	318.3	202.4
A1+A3+A4	539	0.37	164.3	212.7
A2+A3+A4	632	0.49	281.7	431.5
A1+A2+A3+A4	940	0.45	291.7	290.1
Q1+Q2	556	0.18	299.2	313.5
C1+C2+C3	1037	0.5	299.4	395.2
B1	623	0.31	225.4	313.4
W1+W2	608	0.57	295.3	222.1

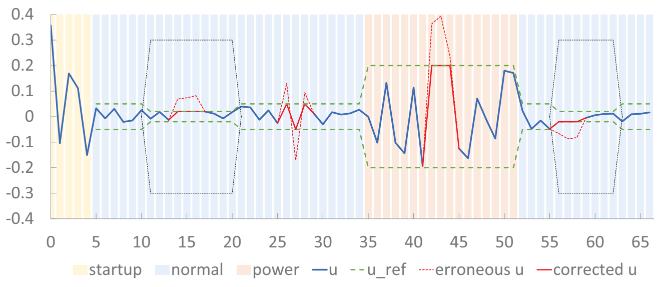

Fig. 7. Automotive powertrain system simulation (w/ and w/o the shield).

Case Study 1: Powertrain Control System To validate the effectiveness of our approach, we integrated the shield into the simulation model of the powertrain control system. Then, we compared the system performance with and without the shield. Fig. 7 shows the simulation results, where our shield was synthesized from the system properties 26, 27, 32, 33 and 34 as described in Jin et al. [19]. Recall that μ is the normalized error of the A/F ratio and μ_{ref} is a reference value.

The green dashed line indicates the safe region, which varies as the system switches between different modes (transition events are highlighted with black dotted line). The red dashed line represents violations of the specification by the O_r signals. The solid red line represents corrections made in O'_r. The result shows that our shield can always produce real-valued correction to keep μ in the safe region.

Recall that the goal of using a shield is not to correct the flawed design $\mathcal{D}$ itself, which includes the overshoot *plant*; instead, the goal is to avoid the negative impact of $\mathcal{D}$'s output. Here, the output signal μ may be used by other components of the system.

Case Study 2: Autonomous Driving Fig. 8 shows the simulation results of an autonomous driving system [35] with and without our shield. Here, an ego vehicle is put into a nondeterministic environment that includes an adversarial vehicle, and the two cars are crossing an intersection. The ego vehicle is protected by a shield synthesized from D1+D2 in Table I. The three plots, from top to bottom, are for distances to the intersection, velocities, and accelerations of the two vehicles. The x-axis represents the time in seconds.

The adversarial vehicle drives straight through the intersection at a constant speed. The ego vehicle, in contrast, may change speed to avoid collision. From $t = 0s$ to $t = 5s$,

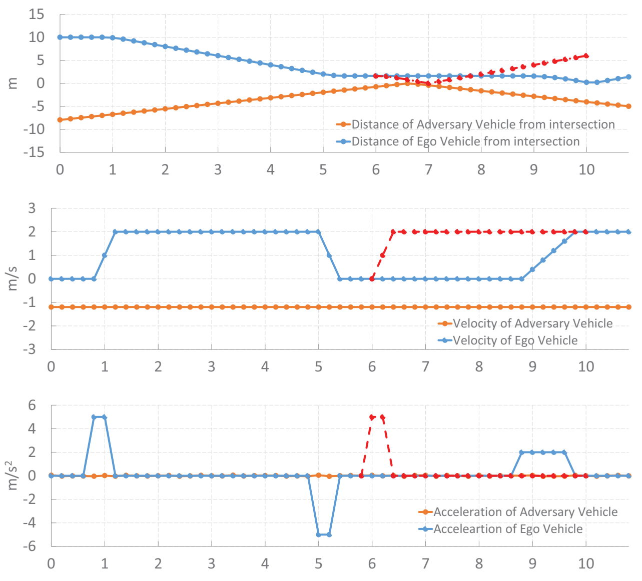

Fig. 8. Position, velocity and acceleration in autonomous driving simulation.

since the distance between the two vehicles is large, the ego vehicle maintains a steady speed (set to $2m/s$ initially). At $t = 5s$, based on the safety specification, it is supposed to come to a stop (for at least $2s$ or when there is no collision risk). However, since we injected an error at $t = 6s$ (in red dashed line), there is an unexpected acceleration and, without the shield, there would have been a collision.

The blue lines show the behavior of the ego vehicle after corrections are made by the shield. Clearly, its behavior satisfies the requirements: it stops at the intersection to allow the adversarial vehicle to pass safely. Furthermore, the real-valued correction made by the shield is successfully predicted using linear regression, and the predicted values satisfy not only the safety but also the robustness requirements.

VII. Related Work

As we have mentioned earlier, prior work on shield synthesis has been restricted to the Boolean domain. Specifically, Bloem et al. [7] introduced the notion of shield together with a synthesis algorithm, which minimizes the deviation between O and O' under the assumption that *no two errors occur within k steps*. Wu et al. [43] improved the algorithm to deal with *burst error*. That is, if more errors occur within the k-step recovery period, instead of entering a *fail-safe* state, they keep minimizing the deviation. Könighofer et al. [22] further improved the shield while Alshiekh et al. [2] leveraged it to improve the performance of reinforcement learning. However, none of the existing techniques dealt with the realizability problems associated with real-valued signals.

There is also a large body of work on reactive synthesis [6], [14], [32], [41] and controller synthesis [17], [24], [34], [35]. The goal is to synthesize $\mathcal{D}$ from a complete specification Ψ, or the control sequences for $\mathcal{D}$ to satisfy Ψ. In both cases, the complexity depends on $\mathcal{D}$. This is a more challenging problem, for two reasons. First, specifying all aspects of the system requirement may be difficult. Second, even if Ψ is available, synthesizing $\mathcal{D}$ from Ψ is difficult due to the inherent *double exponential* complexity of the synthesis problem. Our method, in contrast, treats $\mathcal{D}$ as a blackbox while focusing on a small subset $\varphi \subseteq \Psi$ of *safety-critical* properties. This is why shield synthesis may succeed where conventional reactive synthesis fails.

Renard et al. [36] proposed a runtime enforcement method for timed automata, but assumed that controllable input events may be delayed or suppressed, whereas our method does not require such an assumption. Bauer et al. [4] and Falcone et al. [16] studied various types of temporal logic properties that may be monitored or enforced at run time. Renard et al. [37] also leveraged Büchi games to enforce regular properties with uncontrollable events. Our work is orthogonal in that it tackles the realizability and efficiency problems associated with real-valued signals. Furthermore, we focus on safety while leaving liveness properties and hyper-properties [8], [10], [18] for future work.

An important feature of the shield synthesized by our method is that it always makes corrections *instantaneously*, without any delay. Therefore, it differs from a variety of solutions that allow delayed corrections. In some cases, for example, buffers may be allowed to store the erroneous output temporarily, before computing the corrections [16], [23], [40]. In this context, the notion of *edit-distance* is more relevant. Yu et al. [45], for example, proposed a technique for minimizing the edit-distance between two strings, but the technique requires the entire input be stored in a buffer prior to generating the output. However, when the buffer size reduces to zero, these existing techniques would no longer work.

Runtime enforcement is related to, but also different from, the various software techniques for error avoidance. For example, failure-oblivious computing [25], [38] was used to allow software applications to execute through memory errors; temporal properties [26], [46] were leveraged to control thread schedules to avoid runtime failures of concurrent software. However, these techniques are not designed to target cyber-physical systems with real-valued signals, where corrections are expected to be made instantaneously, i.e., in the same time step when errors occur.

VIII. Conclusions

We have presented a method for synthesizing real-valued shields to enforce the safety of cyber-physical systems. The method relies on a principled technique at the synthesis time to rule out impossible and infeasible scenarios and ensure the realizability of real-valued corrections at run time. We also proposed optimizations to speed up the computation and improve the quality of the solution. We have evaluated our method on a number of applications, including case studies with an automotive powertrain control system and an autonomous driving system. Our results demonstrate the effectiveness of the method in enforcing safety properties of cyber-physical systems.

Acknowledgment

This work was supported by the U.S. National Science Foundation (NSF) under grants CNS-1813117, CNS-1722710, and CCF-1837131.

References

[1] *CUDD: CU Decision Diagram Package*. ftp://vlsi.colorado.edu/pub/.

[2] Mohammed Alshiekh, Roderick Bloem, Rüdiger Ehlers, Bettina Könighofer, Scott Niekum, and Ufuk Topcu. Safe reinforcement learning via shielding. In *AAAI Conference on Artificial Intelligence*, pages 2669–2678, 2018.

[3] Guy Avni, Roderick Bloem, Krishnendu Chatterjee, Thomas A. Henzinger, Bettina Könighofer, and Stefan Pranger. Run-time optimization for learned controllers through quantitative games. In *International Conference on Computer Aided Verification*, pages 630–649, 2019.

[4] Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime verification for LTL and TLTL. *ACM Trans. Softw. Eng. Methodol.*, 20(4):14:1–14:64, 2011.

[5] R. Bloem, K. Chatterjee, K. Greimel, T. Henzinger, G. Hofferek, B. Jobstmann, B. Könighofer, and R. Könighofer. Synthesizing robust systems. *Acta Inf.*, 51:193–220, 2014.

[6] R. Bloem, B. Jobstmann, N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of reactive(1) designs. *J. Comput. Syst. Sci.*, 78(3):911–938, 2012.

[7] Roderick Bloem, Bettina Könighofer, Robert Könighofer, and Chao Wang. Shield synthesis: Runtime enforcement for reactive systems. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, 2015.

[8] Borzoo Bonakdarpour and Bernd Finkbeiner. The complexity of monitoring hyperproperties. In *IEEE Computer Security Foundations Symposium*, pages 162–174, 2018.

[9] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Logics of Programs*, pages 52–71, 1981.

[10] Norine Coenen, Bernd Finkbeiner, César Sánchez, and Leander Tentrup. Verifying hyperliveness. In *International Conference on Computer Aided Verification*, pages 121–139, 2019.

[11] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.

[12] Adel Dokhanchi, Bardh Hoxha, and Georgios E. Fainekos. On-line monitoring for temporal logic robustness. In *International Conference on Runtime Verification*, pages 231–246, 2014.

[13] Tommaso Dreossi, Daniel J. Fremont, Shromona Ghosh, Edward Kim, Hadi Ravanbakhsh, Marcell Vazquez-Chanlatte, and Sanjit A. Seshia. Verifai: A toolkit for the formal design and analysis of artificial intelligence-based systems. In *International Conference on Computer Aided Verification*, pages 432–442, 2019.

[14] Rüdiger Ehlers and Ufuk Topcu. Resilience to intermittent assumption violations in reactive synthesis. In *International Conference on Hybrid Systems: Computation and Control*, pages 203–212, 2014.

[15] Georgios E. Fainekos and George J. Pappas. Robustness of temporal logic specifications. In *International Workshops on Formal Approaches to Software Testing and Runtime Verification*, pages 178–192, 2006.

[16] Yilès Falcone, Jean-Claude Fernandez, and Laurent Mounier. What can you verify and enforce at runtime? *International Journal on Software Tools for Technology Transfer*, 14(3):349–382, 2012.

[17] Samira S Farahani, Vasumathi Raman, and Richard M Murray. Robust model predictive control for signal temporal logic synthesis.

[18] Azadeh Farzan and Anthony Vandikas. Automated hypersafety verification. In *International Conference on Computer Aided Verification*, pages 200–218, 2019.

[19] Xiaoqing Jin, Jyotirmoy V. Deshmukh, James Kapinski, Koichi Ueda, and Ken Butts. Powertrain control verification benchmark. In *International Conference on Hybrid Systems: Computation and Control*, 2014.

[20] Xiaoqing Jin, Alexandre Donzé, Jyotirmoy V Deshmukh, and Sanjit A Seshia. Mining requirements from closed-loop control models. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(11):1704–1717, 2015.

[21] Guy Katz, Derek A. Huang, Duligur Ibeling, Kyle Julian, Christopher Lazarus, Rachel Lim, Parth Shah, Shantanu Thakoor, Haoze Wu, Aleksandar Zeljic, David L. Dill, Mykel J. Kochenderfer, and Clark W. Barrett. The marabou framework for verification and analysis of deep neural networks. In *International Conference on Computer Aided Verification*, pages 443–452, 2019.

[22] Bettina Könighofer, Mohammed Alshiekh, Roderick Bloem, Laura R. Humphrey, Robert Könighofer, Ufuk Topcu, and Chao Wang. Shield synthesis. *Formal Methods in System Design*, 51(2):332–361, 2017.

[23] Jay Ligatti, Lujo Bauer, and David Walker. Run-time enforcement of nonsafety policies. *ACM Trans. Inf. Syst. Secur.*, 12(3), 2009.

[24] Jun Liu, Necmiye Ozay, Ufuk Topcu, and Richard M Murray. Synthesis of switching protocols from temporal logic specifications. 2011.

[25] Fan Long, Stelios Sidiroglou-Douskos, and Martin C. Rinard. Automatic runtime error repair and containment via recovery shepherding. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 227–238, 2014.

[26] Qingzhou Luo and Grigore Roşu. EnforceMOP: a runtime property enforcement system for multithreaded programs. In *International Symposium on Software Testing and Analysis*, pages 156–166, 2013.

[27] Oded Maler and Dejan Nickovic. Monitoring temporal properties of continuous signals. In *Joint International Conferences on Formal Modelling and Analysis of Timed Systems*, pages 152–166, 2004.

[28] Rene Mazala. Infinite games. In *Automata, Logics, and Infinite Games: A Guide to Current Research*, LNCS 2500, 2001.

[29] Petter Nilsson, Omar Hussien, Ayca Balkan, Yuxiao Chen, Aaron D Ames, Jessy W Grizzle, Necmiye Ozay, Huei Peng, and Paulo Tabuada. Correct-by-construction adaptive cruise control: Two approaches. *IEEE Transactions on Control Systems Technology*, 24(4):1294–1307, 2016.

[30] Yash Vardhan Pant, Houssam Abbas, Rhudii A Quaye, and Rahul Mangharam. Fly-by-logic: control of multi-drone fleets with temporal logic objectives. In *ACM/IEEE International Conference on Cyber-Physical Systems*, pages 186–197. IEEE Press, 2018.

[31] A. Pnueli. The temporal logic of programs. In *IEEE Symposium on Foundations of Computer Science*, pages 46–57, 1977.

[32] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 179–190, 1989.

[33] J. P. Quielle and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Symposium on Programming*, 1982.

[34] Vasumathi Raman, Alexandre Donzé, Mehdi Maasoumy, Richard M Murray, Alberto Sangiovanni-Vincentelli, and Sanjit A Seshia. Model predictive control with signal temporal logic specifications. In *IEEE Annual Conference on Decision and Control*, pages 81–87, 2014.

[35] Vasumathi Raman, Alexandre Donzé, Dorsa Sadigh, Richard M Murray, and Sanjit A Seshia. Reactive synthesis from signal temporal logic specifications. In *International Conference on Hybrid Systems: Computation and Control*, pages 239–248. ACM, 2015.

[36] Matthieu Renard, Yliès Falcone, Antoine Rollet, Srinivas Pinisetty, Thierry Jéron, and Hervé Marchand. Enforcement of (timed) properties with uncontrollable events. In *International Colloquium on Theoretical Aspects of Computing*, pages 542–560, 2015.

[37] Matthieu Renard, Antoine Rollet, and Yliès Falcone. Runtime enforcement using büchi games. In *ACM SIGSOFT International SPIN Symposium on Model Checking of Software*, pages 70–79, 2017.

[38] Martin C. Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, Tudor Leu, and William S. Beebee. Enhancing server availability and security through failure-oblivious computing. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 303–316, 2004.

[39] Nima Roohi, Ramneet Kaur, James Weimer, Oleg Sokolsky, and Insup Lee. Parameter invariant monitoring for signal temporal logic. In *International Conference on Hybrid Systems: Computation and Control*, pages 187–196, 2018.

[40] Fred B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3:30–50, 2000.

[41] Saqib Sohail and Fabio Somenzi. Safety first: a two-stage algorithm for the synthesis of reactive systems. *International Journal on Software Tools for Technology Transfer*, 15(5-6):433–454, 2013.

[42] Cumhur Erkan Tuncali, James Kapinski, Hisahiro Ito, and Jyotirmoy V. Deshmukh. Reasoning about safety of learning-enabled components in autonomous cyber-physical systems. In *ACM/IEEE Design Automation Conference*, pages 30:1–30:6, 2018.

[43] Meng Wu, Haibo Zeng, and Chao Wang. Synthesizing runtime enforcer of safety properties under burst error. In *NASA Formal Methods Symposium*, pages 65–81, 2016.

[44] Meng Wu, Haibo Zeng, Chao Wang, and Huafeng Yu. Safety guard: Runtime enforcement for safety-critical cyber-physical systems: Invited. In *ACM/IEEE Design Automation Conference*, pages 84:1–84:6, 2017.

[45] Fang Yu, Muath Alkhalaf, and Tevfik Bultan. Patching vulnerabilities with sanitization synthesis. In *International Conference on Software Engineering*, pages 251–260, 2011.

[46] Lu Zhang and Chao Wang. Runtime prevention of concurrency related type-state violations in multithreaded applications. In *International Symposium on Software Testing and Analysis*, pages 1–12, 2014.