Synthesizing Runtime Enforcer of Safety Specification under Burst Error

Meng Wu, Haibo Zeng, and Chao Wang

Department of ECE, Virginia Tech, Blacksburg, VA 24061, USA

Abstract. We propose a game-based method for synthesizing a runtime enforcer for a reactive system to ensure that a set of safety-critical properties always holds even if errors occur in the system due to design defect or environmental disturbance. The runtime enforcer does not modify the internals of the system or provide a redundant implementation; instead, it monitors the input and output of the system and corrects any erroneous output signal that may cause a safety violation. Our main contribution is a new algorithm for synthesizing a runtime enforcer that can respond to violations *instantaneously* and guarantee the safety of the system *under burst error*. This is in contrast to existing methods that either require significant delay before the enforcer can respond to violations or do not handle burst error. We have implemented our method in a synthesis tool and evaluated it on a set of temporal logic specifications. Our experiments show that the enforcer synthesized by our method can robustly handle a wide range of properties under burst error.

1 Introduction

A reactive system is a system that continuously responds to external events. In practice, reactive systems may have strict timing requirements that demand them to respond without any delay. Furthermore, they are often safety-critical in that a violation may lead to catastrophe. In this context, it is important to guarantee with certainty that the system satisfies a small set of safety properties even in the presence of design defect and environmental disturbance. However, traditional verification and fault-tolerance techniques cannot accomplish this task. In particular, fault-tolerance techniques are not effective in dealing with design defects whereas verification techniques are not effective in dealing with transient faults introduced by the environment. Furthermore, formal verification techniques such as model checking are limited in handling large designs and third-party IP cores without the source code.

In this paper, we propose a new method for synthesizing a runtime enforcer to make sure that a set of safety-critical properties are always satisfied even if the original reactive system occasionally makes mistakes. Unlike the replica in fault-tolerance techniques, our runtime enforcer is significantly cheaper in that it does not attempt to duplicate the functionality of the original system. Instead, it aims at preventing the violation of only a handful of safety properties whose violations may lead to catastrophe. Our approach also differs from classic methods for synthesizing a reactive system itself from the complete specification [14], which is known to be computationally expensive. In our approach, for example, it is perfectly acceptable for the system to violate some

liveness properties, e.g., something good may never happen, as long as it guarantees that safety-critical violations never happen.

The overall flow of our synthesis method is shown in Fig. 1, which takes a safety specification φ^s of the reactive system $\mathcal{D}(I, O)$ as input, and returns another reactive system $\mathcal{S}(I, O, O')$ as output. Following Bloem et al. [3], we call $\mathcal{S}$ the shield. We use I and O to denote the set of input and output signals of the original system, respectively, and define the runtime enforcer $\mathcal{S}(I, O, O')$ as follows: It takes I and O as input and returns a modified version of O as output to guarantee the combined system satisfies the safety specification; that is, $\varphi^s(I, O')$ holds even if $\varphi^s(I, O)$ is violated. Furthermore, the shield modifies O only when $\varphi^s(I, O)$ is violated, and even in that case, it tries to minimize the deviation between O and O'. This approach has several advantages. First, since $\mathcal{S}$ is a reactive system, it can correct the erroneous output in O in the same clock cycle. Second, since $\mathcal{S}$ is agnostic to the size and complexity of the system $\mathcal{D}$, it is cheaper and more scalable than fault-tolerance techniques. Finally, the approach works even if the design contains third-party IP cores.

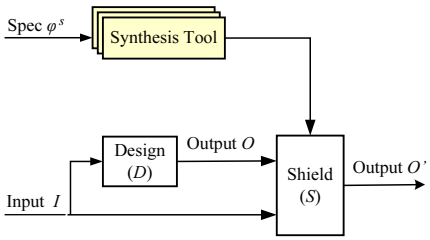

Fig. 1. Synthesizing the safety shield.

Bloem et al. [3] introduced the notion of safety shield and the first algorithm for synthesizing the runtime enforcer, but the method does not robustly handle burst error. Specifically, the shield synthesized by their method minimizes the deviation between O and O' only if no two errors occur within the same k steps. If, for example, another error occurs before the end of this k-step recovery period, the shield would enter the fail-safe state and stop minimizing the deviation. In other words, the shield may generate O' arbitrarily to satisfy $\varphi^s(I, O')$ while ignoring the actual value of O. This often is not the desired behavior, e.g., when the shield enforces mutual-exclusion of a bus arbiter by hard-wiring all output signals to decline all requests.

Our new method, in contrast, can robustly handle burst error. Whenever the design $\mathcal{D}$ satisfies the specification φ^s, our shield ensures that $O' = O$ (no deviation). Whenever $\mathcal{D}$ violates φ^s, our shield takes the best recovery strategy among the set of all possible ones and, unlike the method by Bloem et al. [3], it never enters the fail-safe state. In order words, our method guarantees that the shield $\mathcal{S}$ keeps minimizing the deviation between O to O' even under burst error. We have implemented our new method in a software tool and evaluated it on a range of safety specifications. The experimental results show that the shield synthesized by our method can robustly handle burst error, whereas the shield synthesized by Bloem et al. [3] cannot.

To summarize, this paper makes the following contributions: (1) We propose a new method for synthesizing a runtime enforcer from a set of safety properties that can robustly handle burst error. (2) We implement the method in a software tool and evaluate it on a large set of benchmarks to demonstrate its effectiveness.

The remainder of this paper is organized as follows. First, we illustrate the main ideas of our new method using a motivating example in Section 2. Then, we establish the notation in Section 3 and present our method in Section 4. We develop a technique

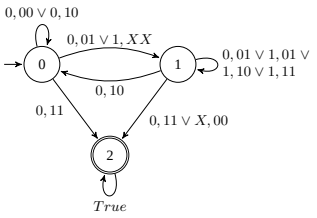

Fig. 2. Example safety specification φ^s.

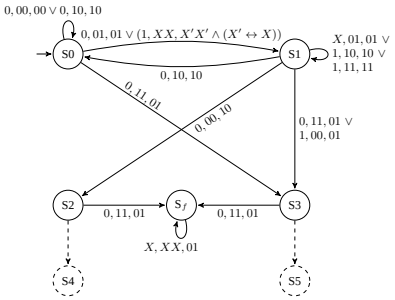

Fig. 3. The 2-stabilizing shield [3].

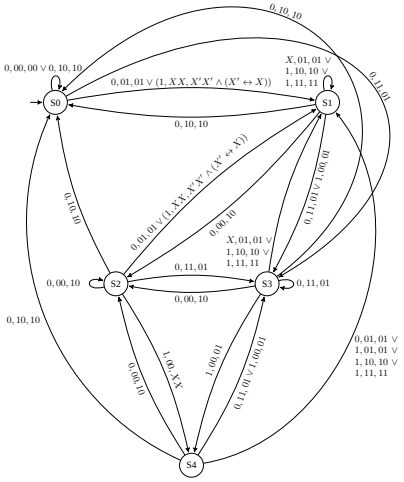

Fig. 4. Our new shield for burst error.

for improving the performance of our synthesis algorithm in Section 5. We describe our experimental results in Section 6. We review the related work in Section 7 and then give our conclusions in Section 8.

2 Motivation

In this section, we use an example to illustrate the main advantage of our shield synthesis method, which is the capability of handling burst error. Consider the automaton representation of a safety specification in Fig. 2, which has three states, one Boolean input signal, and two Boolean output signals. Here, the state 0 is the initial state and the state 2 is the unsafe state. Every edge in the figure represents a state transition. The edge label represents the values of the input and output signals, where the digit before the comma is for the input signal and the two digits after the comma are for the output signals. X stands for *don't care*, meaning that the digit can be either true (1) or false (0). Among other things, the safety specification in Fig. 2 states that when the input value is 0, the two output values cannot be 11; furthermore, in state 1, the two output values cannot be 00.

Assume that the design $\mathcal{D}(i, o_1o_2)$ occasionally violates the safety specification, e.g, by generating 11 for the output signals o_1o_2 when the input i is 0, which forces the automaton to enter the unsafe state. We would like to have the shield $\mathcal{S}(i, o_1o_2, o_1'o_2')$ to produce correct values for the modified output $o_1'o_2'$ as either 10, 01, or 00. Furthermore, whenever the design satisfies the specification or recovers from transient errors, we would like to have the shield produce the same (correct) output as the design; that is, $o_1' = o_1$ and $o_2' = o_2$.

Step	0	1	2	3	4	5	6	7	8	9
Input i	0	0	1	0	0	0	0	0	0	...
Output o_1o_2	00	01	10	11	11	10	10	00	00	...
Shield output $o_1'o_2'$	00	01	10	01	01	01	01	01	01	...
State in Fig. 3	S0	S0	S1	S1	S3	S_f	S_f	S_f	S_f	...

Fig. 5. Simulation trace of 2-stabilizing shield.

Step	0	1	2	3	4	5	6	7	8	9
Input i	0	0	1	0	0	0	0	0	0	...
Design Output o_1o_2	00	01	10	11	11	10	10	00	00	...
Shield output $o_1'o_2'$	00	01	10	01	01	10	10	00	00	...
State in Fig. 4	S0	S0	S1	S1	S3	S3	S0	S0	S0	...

Fig. 6. Simulation trace of our new shield.

Unfortunately, the shield synthesized by Bloem et al. [3] can not always accomplish this task. Indeed, if given the safety specification in Fig. 2 as input, their method would report that a 1-stabilizing shield, which is capable of recovering from a violation in one clock cycle, does not exist, and the best shield their method can synthesize is a 2-stabilizing shield, shown in Fig. 3 (to make it simple, we omit part of the shield unrelated to handle burst error), which requires up to 2 clock cycles to fully recover from a property violation. For example, starting from the initial state $S0$, if the shield sees $i, o_1o_2 = 0, 10$, which satisfies φ^s, it will produce $o_1'o_2' = 10$ and go to the state $S1$. From $S1$, if the shield sees $i, o_1o_2 = 0, 11$, which violates φ^s, it will produce $o_1'o_2' = 01$ and goes to the state $S3$. At this moment, if the second violation $i, o_1o_2 = 0, 11$ occurs, the shield will enter a *fail-safe* state S_f, where it stops minimizing the deviation between $o_1'o_2'$ and o_1o_2.

Fig. 5 shows the simulation trace where two consecutive errors occur in Steps 3 and 4, forcing the shield to enter the fail-safe state s_f where it no longer responds to the original output o_1o_2. This is shown in Steps 5-8, where the original output no longer violates φ^s and yet the shield still modifies the values to 01.

In contrast, our new method would synthesize the shield shown in Fig. 4, which never enters any fail-safe state but instead keeps minimizing the deviation between $o_1'o_2'$ and o_1o_2 even in the presence of burst error. As shown in the simulation trace in Fig. 6, when the two consecutive violations occur in Steps 3 and 4, our new shield will correct the output values to 01. Furthermore, immediately after the design recovers from the transient errors, the shield stops modifying the original output values. Therefore, in Steps 5-8, our shield maintains $o_1'o_2' = o_1o_2$.

3 Preliminaries

In this section, we establish the notation used in the remainder of this paper.

The Reactive System The reactive system to be protected by the shield is represented as a Mealy machine $\mathcal{D} = \langle S, s_0, \Sigma_I, \Sigma_O, \delta, \lambda \rangle$, where S is a finite set of states, $s_0 \in S$ is the initial state, Σ_I is the set of values of the input signals, Σ_O is the set of values of the output signals, δ is the transition function, and λ is the output function. More specifically, $\delta(s, \sigma_I)$ returns the unique next state $s' \in S$ for a given state $s \in S$ and a given input value $\sigma_I \in \Sigma_I$, while $\lambda(s, \sigma_I)$ returns the unique output value $\sigma_O \in \Sigma_O$.

The safety specification that we want to enforce is represented as a finite automaton $\varphi^s = \langle Q, q_0, \Sigma, \delta_\varphi, F_\varphi \rangle$, where Q is a finite set of states, $q_0 \in Q$ is the initial state, $\Sigma = \Sigma_I \times \Sigma_O$ is the input alphabet, δ_φ is the transition function, and $F_\varphi \subseteq Q$ is a set of unsafe (error) states. Let $\overline{\sigma} = \sigma_0\sigma_1 \ldots$ be an input trace where for all $i = 0, 1, \ldots$ we have $\sigma_i \in \Sigma$. Let $\overline{q} = q_0q_1 \ldots$ be the corresponding state sequence such that, for all $i = 0, 1, \ldots$, we have $q_{i+1} = \delta_\varphi(q_i, \sigma_i)$.

We assume the input trace $\overline{\sigma}$ of φ^s is generated by the reactive system $\mathcal{D}$. We say that $\overline{\sigma}$ satisfies φ^s if and only if the corresponding state sequence $\overline{q}$ visits only the safe states; that is, for all $i = 0, 1, \ldots$ we have $q_i \in (Q \setminus F_\varphi)$. We say that $\mathcal{D}$ satisfies φ^s if and only if all input traces generated by $\mathcal{D}$ satisfies φ^s. Let $L(\varphi^s)$ be the set of all input traces satisfying φ^s. Let $L(\mathcal{D})$ be the set of all input traces generated by $\mathcal{D}$. Then, $\mathcal{D}$ satisfies φ^s if and only if $L(\mathcal{D}) \subseteq L(\varphi^s)$.

The Safety Shield Following Bloem et al. [3], we define the shield as another reactive system $\mathcal{S}$ such that, even if $\mathcal{D}$ violates φ^s, the combined system $(\mathcal{D} \circ \mathcal{S})$ still satisfies φ^s. We define the synchronous composition of $\mathcal{D}$ and $\mathcal{S}$ as follows:

Let the shield be $\mathcal{S} = \langle S', s'_0, \Sigma, \Sigma_{O'}, \delta', \lambda' \rangle$, where S' is a finite set of states, $s'_0 \in S'$ is the initial state, $\Sigma = \Sigma_I \times \Sigma_O$ is the input alphabet, $\Sigma_{O'}$, which is the set of values of O', is the output alphabet, $\delta' : S' \times \Sigma \to S'$ is the transition function, and $\lambda' : S' \times \Sigma \to \Sigma_{O'}$ is the output function.

The composition is $\mathcal{D} \circ \mathcal{S} = \langle S'', s''_0, \Sigma_I, \Sigma_{O'}, \delta'', \lambda'' \rangle$, where $S'' = (S \times S')$, $s''_0 = (s_0, s'_0)$, Σ_I is the set of values of the input of $\mathcal{D}$, $\Sigma_{O'}$ is the set of values of the output of $\mathcal{S}$, δ'' is the transition function, and λ'' is the output function. Specifically, $\lambda''((s, s'), \sigma_I)$ is defined as $\lambda'(s', \sigma_I \cdot \lambda(s, \sigma_I))$, which first applies $\lambda(s, \sigma_I)$ to compute the output of $\mathcal{D}$ and then uses $\sigma_I \cdot \lambda(s, \sigma_I)$ as the new input to compute the final output of $\mathcal{S}$. Similarly, δ'' is a combined application of δ and λ from $\mathcal{D}$ and δ' from $\mathcal{S}$. That is, $\delta''((s, s'), \sigma_I) = (\delta(s, \sigma_I), \delta'(s', \sigma_I \cdot \lambda(s, \sigma_I)))$.

Let $L(\mathcal{D} \circ \mathcal{S})$ be the set of input traces generated by the composed system. Clearly, if $L(\mathcal{D}) \subseteq L(\varphi^s)$, the shield $\mathcal{S}$ should simply maintain ${\sigma_O}' = \sigma_O$. But if $L(\mathcal{D}) \nsubseteq L(\varphi^s)$, the shield $\mathcal{S}$ needs to modify the original output of $\mathcal{D}$ to eliminate the erroneous behaviors in $L(\mathcal{D}) \setminus L(\varphi^s)$.

In general, there are multiple ways for $\mathcal{S}$ to change the original output $\sigma_O \in \Sigma_O$ into ${\sigma_O}' \in \Sigma_{O'}$ to eliminate the erroneous behaviors, some of which are better than others in minimizing the deviation. Ideally, we would like the shield to do nothing when $\mathcal{D}$ satisfies φ^s; that is, ${\sigma_O}' = \sigma_O$. However, when $\mathcal{D}$ violates φ^s, the deviation is inevitable. In this case, the shield synthesized by Bloem et al. [3] guarantees that the deviation is minimum only if there are no multiple errors within each k-step recovery period. Under burst error, however, their shield would enter a fail-safe mode where it stops minimizing the deviation. This is undesirable because, even after the transient errors disappear, their shield would still keep modifying the output values.

4 The Synthesis Algorithm

In this section, we present our new shield synthesis algorithm for handling burst error.

4.1 The Overall Flow

Algorithm 1 shows the overall flow of our synthesis procedure. The input of the procedure consists of the safety specification $\varphi^s(I, O)$, and the set of signals in I, O, and O'. The output of the procedure is the safety shield $\mathcal{S}(I, O, O')$.

Starting from the safety specification φ^s, our synthesis procedure first constructs a correctness monitor $\mathcal{Q}(I, O')$. The correctness monitor $\mathcal{Q}$ ensures that the composed

Algorithm 1: Synthesizing the shield $\mathcal{S}(I, O, O')$ from the safety specification $\varphi^s(I, O)$.

```
1: SYNTHESIZE (specification φ^s, input I, output O, modified output O') {
2:   Q(I, O') ← GENCORRECTNESSMONITOR(φ^s)
3:   E(I, O, O') ← GENERRORAVOIDINGMONITOR(φ^s)
4:   G ← Q ∘ E        // create the safety game
5:   ρ ← COMPUTEWINNINGSTREGETY(G)
6:   S(I, O, O') ← CONSTRUCTSHIELD(ρ)
7:   return S
8: }
```

system, whose input is I and output is O', always satisfies the safety specification. That is, $\varphi^s(I, O')$ holds even if $\varphi^s(I, O)$ occasionally fails. Note that $\mathcal{Q}(I, O')$ alone may not be sufficient as a specification for synthesizing the desired shield $\mathcal{S}$, because it refers only to O' but not to O. For example, if we give $\mathcal{Q}$ to a classic reactive synthesis procedure, e.g., Pnueli and Rosner [14], it may produce a shield that ignores the original output O of the design and arbitrarily generates O' to satisfy $\varphi^s(I, O')$.

To minimize the deviation from O to O', we construct an error-avoiding monitor $\mathcal{E}(I, O, O')$ from φ^s. In this work, we use the Hamming distance between O and O' as the measurement of the deviation. Therefore, when the design $\mathcal{D}(I, O)$ satisfies $\varphi^s(I, O)$, the error-avoiding monitor ensure that $O' = O$. When $\mathcal{D}(I, O)$ violates $\varphi^s(I, O)$, however, we have to modify the output to avoid the violation of $\varphi^s(I, O')$; in such cases, we want to impose constraints in $\mathcal{E}$ so as to minimize the deviation from O to O'. The detailed algorithm for constructing $\mathcal{E}$ is presented in Section 4.2. Essentially, $\mathcal{E}(I, O, O')$ captures all possible ways of modifying O to O' to minimize the deviation. To pick the best possible modification strategy, we formulate the synthesis problem as a two-player safety game, where the shield corresponds to a winning strategy. Toward this end, we define a set of *unsafe* states of $\mathcal{E}$ as follows: they are the states where $\varphi^s(I, O)$ holds but $O' \neq O$, and they must be avoided by the shield while it modifies O to O'.

The two-player safety game is played in the game graph $\mathcal{G} = \mathcal{Q} \circ \mathcal{E}$, which is a synchronous composition of the correctness monitor $\mathcal{Q}$ and the error-avoiding monitor $\mathcal{E}$. Recall that $\mathcal{Q}$ is used to make sure that $\varphi^s(I, O')$ holds, and $\mathcal{E}$ is used to make sure that $O' = O$ whenever $\varphi^s(I, O)$ holds. Therefore, the set of *unsafe* states of $\mathcal{G}$ is defined as follows: they are the states that are unsafe in either $\mathcal{Q}$ or $\mathcal{E}$. Conversely, the *safe* states of $\mathcal{G}$ are those that simultaneously guarantee $\varphi^s(I, O')$ and minimum deviation from O to O'. The main difference between our new synthesis method and the method of Bloem et al. [3] is in the construction of this safety game: their method does not allow the second error to occur in O during the k-step recovery period of the first error, whereas our new method allows such error.

After solving the two-player safety game denoted as $\mathcal{G}(I, O, O')$, we obtain a winning strategy $\rho = (\delta_\rho, \lambda_\rho)$, which allows us to stay in the safe states of $\mathcal{G}$ by choosing proper values of O' regardless of the values of I and O. The winning strategy consists of two parts: δ_ρ is the transition function that takes a present state of $\mathcal{G}$ and values of I and O as input and returns a new state of $\mathcal{G}$, and λ_ρ is the output function that takes a present state of $\mathcal{G}$ and values of I and O as input and returns a new value for O'. Finally, we convert the winning strategy ρ into the shield $\mathcal{S}$, which is a reactive system that implements the transition function and output function in ρ.

Fig. 7. Example: (a) safety specification $\varphi^s(R, S)$ and (b) correctness monitor $\mathcal{Q}(R, S')$.

4.2 Constructing the Safety Game

We first use an example to illustrate the construction of the safety game $\mathcal{G}$ from φ^s. Consider Fig. 7 (a), which shows the automaton representation of a safety property of the ARM bus arbiter [2]; the LTL formula is $\mathsf{G}(\neg R \rightarrow \mathsf{X}(\neg S))$, meaning that transmission cannot be *started* (S is the output) if the bus is not *ready* (R is the input signal). In Fig. 7 (a), the state 2 is unsafe. The first step of our synthesis procedure is to construct the correctness monitor $\mathcal{Q}(R, S')$, shown in Fig. 7 (b), which is a duplication of $\varphi^s(R, S)$ except for replacing the original output S with the modified output S'.

The next step is to construct the error-avoiding monitor $\mathcal{E}(R, S, S')$, which captures all possible ways of modifying S into S' to avoid reaching the unsafe state. This is where our method differs from Bloem et al. [3] the most. Specifically, Bloem et al. [3] assume that the second violation from the design will not occur during the k-step recovery period of the first violation. If there are more than one violations within k steps, it would enter a *fail-safe* state S_f, where it stops tracking the deviation from S to S'. Our method, in contrast, never enters the *fail-safe* state. It starts from the safety specification φ^s and replaces all transitions to the *unsafe* state with transitions to some safe states. This is achieved by modifying the value of the output signal S so that the transition matches some existing transition to a safe state. If there are multiple ways of modifying S to redirect the edges leading to unsafe states in φ^s, we simultaneously track all of these choices until the ambiguity is completely resolved. In other words, we keep correcting consecutive violations without ever giving up (entering S_f). This is done by modifying the error tracking automaton which is responsible for motoring the behavior of design: we conservatively assume the design will make mistakes at any time, so whenever there is a chance for the design to make mistakes, we generate a new abstract state to guess its correct behaviors.

Construction of $\mathcal{E}(I, O, O')$ Algorithm 2 shows the pseudocode for constructing the error-avoiding monitor $\mathcal{E}$. At the high level, $\mathcal{E} = \mathcal{U} \circ \mathcal{T}$, where $\mathcal{U}(I, O)$ is called the violation monitor and $\mathcal{T}(O, O')$ is called the deviation monitor.

– To construct the violation monitor $\mathcal{U}$, we start with a copy of the specification automaton φ^k, and then replace each existing edge to a failing state, denoted as $(s, l) \rightarrow t$, with an edge to a newly added abstract state s_g, denoted as $(s, l) \rightarrow s_g$. The abstract state s_g represents the set of possible safe states to which we may redirect the erroneous edge. That is, each safe state $s' \in s_g.states$ may be reached from s through $(s, l') \rightarrow t'$, where l, l' share common input label. Since each guessing state s_g represents a subset of the safe states in φ^s, the procedure for constructing $\mathcal{U}(I, O)$ from $\varphi^s(I, O)$ resembles the classic procedure for subset construction.

Algorithm 2: Generating error-avoiding monitor $\mathcal{E}$ from safety specification φ^s.

```
1:  GENERRORAVOIDINGMONITOR ( specification φ^s ) {
2:    U ← copy of the specification automaton φ^s
3:    while (∃ edge (s, l) → t in U where t is an unsafe state) {
4:      Delete edge (s, l) → t from U
5:      Add abstract state s_g and edge (s, l) → s_g into U    //{t'} ⊆ s_g.states
6:      foreach (edge (s, l') → t' such that t' is safe, and l, l' share common input)
7:        foreach (outgoing edge (t', l'') → t'')
8:          Add edge (s_g, l'') → t'' into U
9:      U ← MERGEEDGESWITHSAMELABEL(U)
10:   }
11:   T ← the deviation monitor
12:   E ← U ∘ T
13:   return E
14: }
15: MERGEEDGESWITHSAMELABEL(monitor U) {
16:   while (∃ edges (s_g, l_1) → t_1 and (s_g, l_2) → t_2 in U where l_1 ∧ l_2 is not false) {
17:     Delete edges (s_g, l_1) → t_1 and (s_g, l_2) → t_2 from U
18:     if (l_1 ∧ ¬l_2 is not false)    Add edge (s_g, l_1 ∧ ¬l_2) → t_1 back to U
19:     if (l_2 ∧ ¬l_1 is not false)    Add edge (s_g, l_2 ∧ ¬l_1) → t_2 back to U
20:     Add abstract state s_m and edge (s_g, l_1 ∧ l_2) → s_m to U    //{t_1, t_2} ⊆ s_m.states
21:     foreach (outgoing edge of t_1 and t_2, denoted as (t_12, l') → t')
22:       Add edge (s_m, l') → t' into U
23:     if (t_1 or t_2 is unsafe)    return U
24:   }
25: }
```

– To construct the deviation monitor $\mathcal{T}$, we start by creating two states A and B and treating values of O and O' as the input symbols. Whenever $O = O'$, the state transition goes to state A, and whenever $O \neq O'$, the state transition goes to B. Finally, we label A as the safe state and B as the unsafe state. Fig. 10 shows the deviation monitor.

Consider the safety specification $\varphi^s(R, S)$ in Fig. 7 (a) again. To construct the violation monitor $\mathcal{U}(R, S)$, we first make a copy of the automaton φ^s, as shown in Line 2 of Algorithm 2. Then, starting from Line 3, we replace the edge to the unsafe state 2, denoted as $(1, S) \rightarrow 2$, with the edge to a guessing state, denoted as $(1, S) \rightarrow 2_g$, where the set of safe states in 2_g is $\{0, 1\}$. That is, if we modify the output value S to the new value $\neg S$, the transition from state 1 may go to either state 0 or state 1. This is shown in Fig. 8 (a). In Lines 6-8, for each outgoing edge of the states in $\{0, 1\}$, we add an outgoing edge from 2_g.

Next, we merge the outgoing edges with the same label in Line 9. This acts like a subset construction. For example we may first merge two edges with the label $R \wedge \neg S$, both of them lead to state 0. Then, we merge the two edges with the label $\neg R \wedge \neg S$. Then, consider the edge label $\neg R \wedge S$: starting from state $0 \in 2_g$, the next state is 1, and starting from state $1 \in 2_g$, the next state is 2. Therefore, the outgoing edge labeled $\neg R \wedge S$ goes to the abstract state 4_m, whose set of states is $\{1, 2\}$. Since 2 is an unsafe state, we return back to Line 3 in Algorithm 2 and replace it with other guessing states.

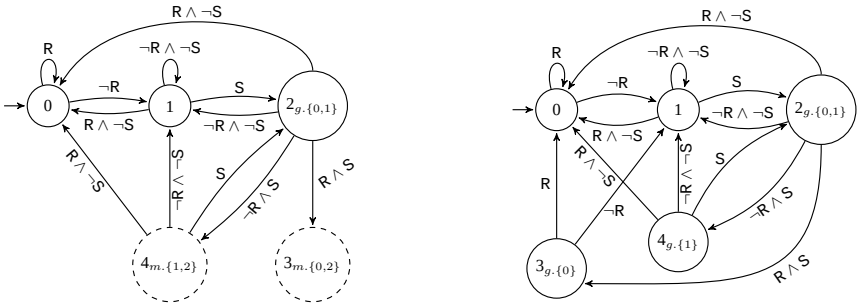

Fig. 8. Constructing the violation monitor $\mathcal{U}(R, S)$: Replacing edge $1 \to 2$ with $1 \to \{0, 1\}$.

More specifically, the state 2 is replaced with the state 1 and 4_m becomes 4_g. After adding all outgoing edges of 4_g, the resulting $\mathcal{U}$ is shown in Fig. 8 (a). Similarly, we merge the remaining outgoing edges of 2_g that are labeled $R \wedge S$ and create the abstract state 3_m, whose set of states is $\{0, 2\}$. Since 2 is an unsafe state, we go back to Line 3 and replace it again. This turns 3_m into 3_g and the resulting automaton is shown in Fig. 8 (b). At this moment, all error states (state 2) are eliminated and therefore $\mathcal{U}$ is fully constructed.

Unsafe States of $\mathcal{E} = \mathcal{U} \circ \mathcal{T}$ The error-avoiding monitor $\mathcal{E}$ is a synchronous composition of $\mathcal{U}$ and $\mathcal{T}$, where the unsafe states are defined as the union of the following sets:

- $\{(s, B) \mid s$ is a safe state in $\mathcal{U}$ coming from $\varphi^s\}$,
- $\{(s_m, B) \mid s_m$ results from merging edges and it contains no unsafe state$\}$, and
- $\{(s_g, A) \mid s_g$ results from replacing some unsafe states$\}$.

The reason is, when s is a safe state and s_m contains only safe states, the specification φ^s is not violated and therefore we must ensure $O' = O$ (state A in $\mathcal{T}$). In contrast, since s_g is created by replacing some originally unsafe states, the specification $\varphi^s(I, O)$ is violated, in which case $O' \neq O$ in order to avoid the violation of $\varphi^s(I, O')$. Figs. 9-11 show the resulting error-avoiding automaton. For brevity, only safe states and edges among these states are shown in Fig. 11. Note that 2_gB, 3_gB, 4_gB are there because they are created by replacing some unsafe states and $O' \neq O$ holds in the B states.

Fig. 12 shows the game graph $\mathcal{G} = \mathcal{Q} \circ \mathcal{E}$ for the correctness monitor $\mathcal{Q}$ in Fig. 7 (b) and the error-avoiding monitor $\mathcal{E}$ in Fig. 11. For brevity, only the safe states in $\mathcal{G}$ and edges among these states are shown in Fig. 12. A safe state in $\mathcal{G}$ is a state $(g_\mathcal{Q}, g_\mathcal{E})$ where $g_\mathcal{Q}$ is safe in $\mathcal{Q}$ and $g_\mathcal{E}$ is safe in $\mathcal{E}$. The winning strategy of this safety game is denoted as $\rho = (\delta_\rho, \lambda_\rho)$, where δ_ρ is the transition function capturing a subset of the edges in Fig. 12, and λ_ρ is the output function determining the value of S' based on the current state and values of R and S. The shield $\mathcal{S}(R, S, S')$ is a reactive system that implements function δ_ρ and λ_ρ of ρ.

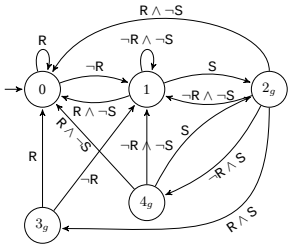

Fig. 9. Violation monitor $\mathcal{U}(R, S)$.

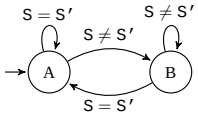

Fig. 10. Deviation monitor $\mathcal{T}(S, S')$.

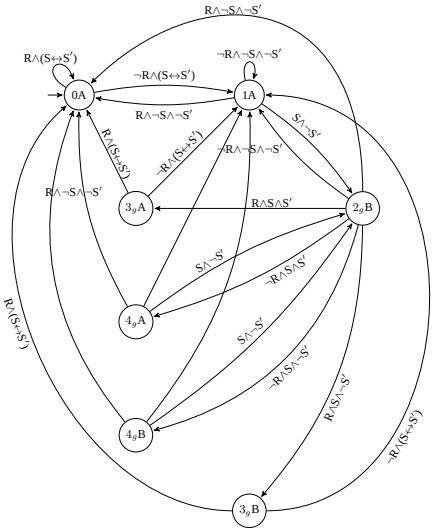

Fig. 11. Error-avoiding monitor $\mathcal{E}(R, S, S')$.

5 Solving the Safety Game

We compute the winning strategy $\rho = (\delta_\rho, \lambda_\rho)$ by solving the two-player safety game $\mathcal{G} = (G, g_0, \Sigma, \Sigma_{O'}, \delta, F)$, where G is a finite set of game states, $g_0 \in G$ is the initial state, $F \subseteq G$ are the final (unsafe) states, $\delta : G \times \Sigma \times \Sigma_{O'} \to G$ is a complete transition function. The two players of the game are the shield and the environment (including the design $\mathcal{D}$). In every game state $g \in G$, the environment first chooses an input letter $\sigma \in \Sigma$, and then the shield chooses some output letter $\sigma_O{}' \in \Sigma_{O'}$, leading to the next state $g' = \delta(g, \sigma, \sigma_O{}')$. The sequence $\overline{g} = g_0 g_1 \ldots$ of game states is called a *play*. We say that a play is *won* by the shield if and only if, for all $i = 0, 1, \ldots$ we have $g_i \in G \setminus F$.

5.1 Fix-point Computation

In this work, we use the algorithm of Mazala [12] to solve the safety game. In this algorithm, we compute "attractors" for a subset of safe states $(G \setminus F)$ and final states (F), until reaching the fix-point. Specifically, we maintain two sets of states: $\mathcal{F}$ is the set of states from which the shield will inevitably lose, and $\mathcal{W}$ is the set of states from which the shield has a strategy to win. We also define a function

$$MX(Z) = \{q \mid \exists \sigma \in \Sigma \,.\, \forall \sigma_O{}' \in \Sigma_{O'} \,.\, q' = \delta(q, \sigma, \sigma_O{}') \wedge (q' \in Z)\}$$

That is, $MX(Z)$ is the set of states from which the environment can force the transition to a state in Z regardless of how the shield responds.

The fix-point computation starts with $\mathcal{W} = G \setminus F$ and $\mathcal{F} = F$. In each iteration, $\mathcal{W} = \mathcal{W} \setminus MX(\mathcal{F})$ and $\mathcal{F} = \mathcal{F} \cup MX(\mathcal{F})$. The computation stops when both $\mathcal{W}$ and $\mathcal{F}$ reach the fix-point.

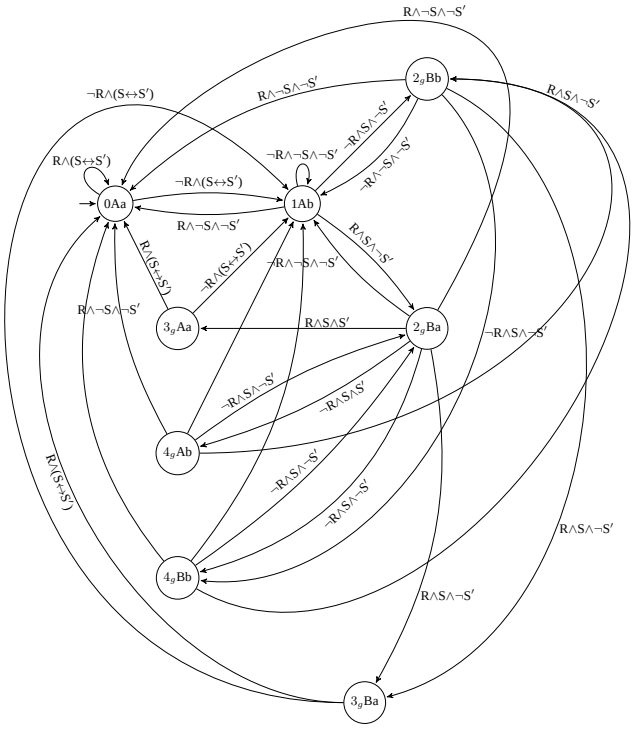

Fig. 12. The game graph $\mathcal{G}(R, S, S')$, which is the composition of $\mathcal{Q}(R, S')$ and $\mathcal{E}(R, S, S')$.

5.2 Optimization

The computation of the winning strategy ρ in the safety game $\mathcal{G} = \mathcal{E} \circ \mathcal{Q}$ is time-consuming. In this section, we propose a new method for speeding up this computation. First, we note that a safe state in $\mathcal{G}$ must be safe in both $\mathcal{E}$ and $\mathcal{Q}$, meaning that a winning play in $\mathcal{G}$ must be winning in both of the subgames $\mathcal{E}$ and $\mathcal{Q}$. Therefore, instead of directly computing the winning region $\mathcal{W}$ of $\mathcal{G}$, which can be expensive due to the size of $\mathcal{G}$, we first compute the winning region $\mathcal{W}_1$ of the smaller subgame $\mathcal{G}_1 = \mathcal{E}$, then compute the winning region $\mathcal{W}_2$ of the smaller subgame $\mathcal{G}_2 = \mathcal{Q}$, and finally compute the winning region $\mathcal{W}$ of the game $\mathcal{G}$ by using $\mathcal{W}_1 \times \mathcal{W}_2$ as the starting point. Since a winning play in $\mathcal{G}$ is winning in both $\mathcal{G}_1$ and $\mathcal{G}_2$, we know $\mathcal{W} \subseteq \mathcal{W}_1 \times \mathcal{W}_2$.

Furthermore, due to the unique characteristics of the subgames $\mathcal{G}_1 = \mathcal{E}$ and $\mathcal{G}_2 = \mathcal{Q}$, in practice, $\mathcal{W}_1 \times \mathcal{W}_2$ is often close to the final fix-point $\mathcal{W}$. This is because both $\mathcal{E}(I, O, O')$ and $\mathcal{Q}(I, O')$ are derived from the specification automaton φ^s. Specifically, each state in $\mathcal{Q}$ is simply a copy of the corresponding state in φ^s, whereas each state in $\mathcal{E}$ is either a copy of a safe state s in φ^s, or a new abstract state s_g that replaces some unsafe states in φ^s, or a new abstract state s_m consisting of only safe states in φ^s. Since it is cheaper to compute $\mathcal{W}_1$ and $\mathcal{W}_2$, this optimization can significantly speed up the fix-point computation.

6 Experiments

We have implemented our new method in the same software tool that also implements the method of Bloem et al. [3]. The fix-point computation for solving safety games is implemented symbolically, using CUDD [18] as the BDD library, whereas the construction of the various monitors and the game graph are carried out explicitly. The tool takes the automaton representation of the safety specification φ^s as input and returns the Verilog program of the synthesized shield $\mathcal{S}$ as output.

We have evaluated our method on a range of safety specifications, including temporal logic properties from (1) the Toyota powertrain control verification benchmark [9], (2) an automotive design for engine and brake controls [13], (3) the traffic light controller example from the VIS model checker [4], (4) LTL property specification patterns from Dwyer et al. [6], and (5) parts of the ARM AMBA bus arbiter specification [2]. Specifically, properties from [9] are on the model of a fuel control system, specifying the performance requirements in various operation modes. Originally, they were represented in signal temporal logic (STL). We translated them to LTL by replacing the predicates over real variables with boolean variables. The properties for engine and brake control [13] are related to the safety of the brake overriding mechanism. The properties for traffic light controller [4] are for safety of a crossroads traffic light. The AMBA benchmark [2] includes combinations of various properties of an ARM bus arbiter. We also translate liveness properties in Dwyer et al. [6] to safety properties by adding a bound on the reaction time steps. For example, in the first columns of Table 1, the numbers besides F and U are the bound number, where F and U mean *Finally* and *Until* respectively. Details of these benchmarks can be found in the supplementary document on our tool repository website [19].

Table 1 shows the results of running our tool on these benchmarks and comparing it with the method of Bloem et al. [3]. Columns 1-2 show the benchmark name and the number of states of the safety specification φ^s. Columns 3-5 show the results of applying the k-stabilizing shield synthesis algorithm [3], including whether the resulting shield can handle burst error, the shield size in terms of the number of states, and the synthesis time in seconds. Similarly, Columns 6-8 show the results of applying our new synthesis algorithm. Note that the k-stabilizing shields do not guarantee to handle burst error, and as shown in Table 1, only some of them can actually handle burst error. Here, "*no (1-step)*" means the shield needs at least one more clock cycle to recover from the previous error before it can take on the next error, and "*no (2-step)*" means the shield needs at least two more clock cycles to recover. In contrast, the shield synthesized by our new method can recover instantaneously and therefore can always handle burst error.

In terms of the synthesis time, the result is mixed in that our new method is sometimes slower and sometimes faster than the existing method. There are two reasons for such results. On the one hand, our method is searching through a significantly larger game graph than the existing method in order to find the best winning strategy for handling burst error. On the other hand, our method utilizes the new optimization technique described in Section 5.2 for symbolically computing the winning region, which can significantly speed up the fix-point computation.

Table 2 shows the results of our synthesis algorithm with and without optimization. Columns 1-2 show the benchmark name and the size of the safety specification.

Table 1. Experimental results for comparing the two shield synthesis algorithms.

Property φ^s	Sates	K-Stabilizing Shield [3]			Burst-Error Shield (New)		
		Handle-Burst-Error	States in $\mathcal{S}$	Time (s)	Handle-Burst-Error	States in $\mathcal{S}$	Time (s)
Toyota powertrain [9]	23	yes	38	0.2	yes	38	0.3
Engine and brake ctrl [13]	5	yes	7	0.1	yes	7	0.1
Traffic light [4]	4	yes	7	0.1	yes	7	0.2
$\mathsf{F}_{64}\, p$ [6]	67	yes	67	0.7	yes	67	0.5
$\mathsf{F}_{256}\, p$	259	yes	259	46.9	yes	259	10.5
$\mathsf{F}_{512}\, p$	515	yes	515	509.1	yes	515	54.4
$\mathsf{G}(\neg q) \vee \mathsf{F}_{64}(q \wedge \mathsf{F}_{64}\, p)$ [6]	67	yes	67	0.7	yes	67	0.6
$\mathsf{G}(\neg q) \vee \mathsf{F}_{256}(q \wedge \mathsf{F}_{256}\, p)$	259	yes	259	46.9	yes	259	10.7
$\mathsf{G}(\neg q) \vee \mathsf{F}_{512}(q \wedge \mathsf{F}_{512}\, p)$	515	yes	515	517.7	yes	515	54.5
$\mathsf{G}(q \wedge \neg r \rightarrow (\neg r\, \mathsf{U}_4(p \wedge \neg r)))$ [6]	6	yes	15	0.1	yes	145	0.1
$\mathsf{G}(q \wedge \neg r \rightarrow (\neg r\, \mathsf{U}_8(p \wedge \neg r)))$	10	yes	109	0.2	yes	5,519	4.5
$\mathsf{G}(q \wedge \neg r \rightarrow (\neg r\, \mathsf{U}_{12}(p \wedge \neg r)))$	14	yes	753	6.3	yes	27,338	1,414.5
AMBA G1+2+3 [2]	12	yes	22	0.1	yes	22	0.1
AMBA G1+2+4 [2]	8	no (1-step)	61	6.3	yes	78	2.2
AMBA G1+3+4 [2]	15	no (1-step)	231	55.6	yes	640	97.6
AMBA G1+2+3+5 [2]	18	no (1-step)	370	191.8	yes	1,405	61.8
AMBA G1+2+4+5 [2]	12	no (1-step)	101	3,992.9	yes	253	472.9
AMBA G4+5+6 [2]	26	no (2-step)	252	117.9	yes	205	26.4
AMBA G5+6+10 [2]	31	no (2-step)	329	9.8	yes	396	31.4
AMBA G5+6+9e4+10 [2]	50	no (2-step)	455	17.6	yes	804	42.1
AMBA G5+6+9e8+10 [2]	68	no (2-step)	739	34.9	yes	1,349	86.8
AMBA G5+6+9e16+10 [2]	104	no (2-step)	1,293	74.7	yes	2,420	189.7
AMBA G5+6+9e64+10 [2]	320	no (2-step)	4,648	1,080.8	yes	9,174	2,182.5
AMBA G8+9e4+10 [2]	48	no (2-step)	204	7.0	yes	254	6.1
AMBA G8+9e8+10 [2]	84	no (2-step)	422	22.5	yes	685	33.7
AMBA G8+9e16+10 [2]	156	no (2-step)	830	83.7	yes	1,736	103.1
AMBA G8+9e64+10 [2]	588	no (2-step)	3,278	2,274.2	yes	7,859	2,271.5

Columns 3-4 show the size of the resulting shield and the synthesis time without using the optimization. Columns 5-6 show the shield size and the synthesis time with the optimization. In almost all cases, there is significant reduction in the synthesis time when the optimization is used. At the same time, there is slightly difference in the number of states in the resulting shield. This is because the game graph often contains multiple winning strategies, and currently our method for computing the winning strategy tends to pick an arbitrary one. Furthermore, since the shield is implemented in hardware, the difference in the number of bit-registers (flip-flops) needed to implement the two shields will be further reduced. For example, in the last benchmark, we have $\lceil log_2(3278) \rceil = 12$, whereas $\lceil log_2(7859) \rceil = 13$, meaning that the shield requires either 12 or 13 bit-registers. Nevertheless, for future work, we plan to investigate new ways of computing the winning strategy to further reduce the shield size.

7 Related Work

As we have already mentioned, our method for ensuring that the design $\mathcal{D}$ always satisfies the safety specification φ^s differs from both model checking [5, 15], which checks whether $\mathcal{D} \models \varphi^s$ but does not enforce φ^s, and reactive synthesis [14, 2, 7], which synthesizes the design $\mathcal{D}$ from a complete specification. Since our method is agnostic to the size and complexity of $\mathcal{D}$, it can be significantly more scalable than reactive synthesis in practice. Our method differs from the existing shield synthesis method of Bloem et al. [3] in that it can robustly handle burst error.

Table 2. Experimental results for synthesizing the shield with and without optimization.

Property φ^s	States	Burst Error Shield Syn. (w/o Opt)		Burst Error Shield Syn. (w/ Opt)	
		States in $\mathcal{S}$	Time (s)	States in $\mathcal{S}$	Time (s)
Toyota powertrain [9]	23	38	0.3	38	0.3
Engine and brake ctrl [13]	5	7	0.1	7	0.1
Traffic light [4]	4	7	0.2	7	0.2
$\mathsf{F}_{64}\, p$ [6]	67	67	0.7	67	0.5
$\mathsf{F}_{256}\, p$	259	259	45.5	259	10.5
$\mathsf{F}_{512}\, p$	515	5157	511.0	515	54.4
$\mathsf{G}(\neg q) \vee \mathsf{F}_{64}(q \wedge \mathsf{F}_{64}\, p)$ [6]	67	67	0.8	67	0.6
$\mathsf{G}(\neg q) \vee \mathsf{F}_{256}(q \wedge \mathsf{F}_{256}\, p)$	259	259	46.2	259	10.7
$\mathsf{G}(\neg q) \vee \mathsf{F}_{512}(q \wedge \mathsf{F}_{512}\, p)$	515	515	668.1	515	54.5
$\mathsf{G}(q \wedge \neg r \rightarrow (\neg r\, \mathsf{U}_4(p \wedge \neg r)))$ [6]	6	98	0.1	145	0.1
$\mathsf{G}(q \wedge \neg r \rightarrow (\neg r\, \mathsf{U}_8(p \wedge \neg r)))$	10	4,002	3.9	5,519	4.5
$\mathsf{G}(q \wedge \neg r \rightarrow (\neg r\, \mathsf{U}_{12}(p \wedge \neg r)))$	14	95,357	1,506.9	27,338	1,414.5
AMBA G1+2+3 [2]	12	22	0.1	22	0.1
AMBA G1+2+4 [2]	8	69	2.3	78	2.2
AMBA G1+3+4 [2]	15	566	99.5	640	97.6
AMBA G1+2+3+5 [2]	18	1,256	58.4	1,405	61.8
AMBA G1+2+4+5 [2]	12	193	479.2	253	472.9
AMBA G4+5+6 [2]	26	206	26.3	205	26.4
AMBA G5+6+10 [2]	31	413	30.5	396	31.4
AMBA G5+6+9e4+10 [2]	50	796	40.4	804	42.1
AMBA G5+6+9e8+10 [2]	68	1,287	80.8	1,349	86.8
AMBA G5+6+9e16+10 [2]	104	2,334	194.2	2,420	189.7
AMBA G5+6+9e64+10 [2]	320	8,618	2,865.6	9,174	2,182.5
AMBA G8+9e4+10 [2]	48	233	5.6	254	6.13
AMBA G8+9e8+10 [2]	84	601	30.5	685	33.7
AMBA G8+9e16+10 [2]	156	1,344	111.0	1,736	103.1
AMBA G8+9e64+10 [2]	588	5,848	7,843	7,859	2,271.5

Our shield is a reactive system that can respond to a safety violation instantaneously, e.g., in the same clock cycle where the violation occurs, and therefore differs from the many existing methods for enforcing temporal properties [17, 10, 8] that have to buffer the erroneous output before correcting them. Similarly, it differs from the method by Luo and Rosu [11] for enforcing temporal logic properties in concurrent software, which relies on delaying the execution of one or more threads to avoid unsafe states. It also differs from the method by Yu et al. [20], which aims at minimizing the edit-distance between two strings, but requires the entire input string to be available prior to generating the output string.

Renard et al. [16] proposed a runtime enforcement method for timed-automaton properties, but the method differs from ours as it assumes that the controllable input events can be delayed or suppressed, whereas our method relaxes such an assumption. Bauer et al. [1] and Falcone et al. [8] also studied what type of temporal logic properties can or cannot be monitored and enforced at run time. These works are orthogonal and complementary to ours. In this work, we focus on enforcing safety specification only. We leave the enforcement of liveness properties for future work.

8 Conclusions

We have presented a new method for synthesizing a runtime enforcer to ensure that a small set of safety-critical properties always hold in a reactive system. The shield responds to property violations instantaneously and robustly handles burst error. We

have also presented an optimization technique for speeding up the symbolic fix-point computation for solving the underlying safety games. We have implemented our method in a software tool and evaluated it on a set of benchmarks. Our experimental results show that the new method is significantly more effective than existing methods for handling burst error.

References

1. A. Bauer, M. Leucker, and C. Schallhart. Runtime verification for LTL and TLTL. *ACM Trans. Softw. Eng. Methodol.*, 20(4):14:1–14:64, September 2011.
2. 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.
3. R. Bloem, B. Könighofer, R. Könighofer, and C. Wang. Shield synthesis: Runtime enforcement for reactive systems. In *Intl. Conf. Tools and Algorithms for Construction and Analysis of Systems*, 2015.
4. R. K. Brayton et al. VIS: A system for verification and synthesis. In *Intl. Conf. Computer Aided Verification*, pages 428–432, 1996.
5. E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Logics of Programs*, LNCS 131, pages 52–71, 1981.
6. M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *Intl. Conf. Software Engineering*, 1999.
7. R. Ehlers and U. Topcu. Resilience to intermittent assumption violations in reactive synthesis. In *Intl. Conf. Hybrid Systems: Computation and Control*, pages 203–212, 2014.
8. Y. Falcone, J.-C. Fernandez, and L. Mounier. What can you verify and enforce at runtime? *J. Software Tools for Technology Transfer*, 14(3):349–382, 2012.
9. X. Jin, J. Deshmukh, J. Kapinski, K. Ueda, and K. Butts. Powertrain control verification benchmark. In *Intl. Conf. Hybrid Systems: Computation and Control*, 2014.
10. J. Ligatti, L. Bauer, and D. Walker. Run-time enforcement of nonsafety policies. *ACM Trans. Inf. Syst. Secur.*, 12(3), 2009.
11. Q. Luo and G. Roşu. Enforcemop: a runtime property enforcement system for multithreaded programs. In *Intl. Symp. Software Testing and Analysis*, pages 156–166, 2013.
12. R. Mazala. Infinite games. In *Automata, Logics, and Infinite Games: A Guide to Current Research*, LNCS 2500, 2001.
13. NHTSA. *49 CFR Part 571: Federal Motor Vehicle Safety Standards; Accelerator Control Systems*. Department of Transportation, 2012.
14. A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *ACM Symp. Principles of Programming Languages*, pages 179–190, 1989.
15. J. P. Quielle and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Symposium on Programming*, LNCS 137. Springer, 1982.
16. M. Renard, Y. Falcone, and A. Rollet. Optimal enforcement of (timed) properties with uncontrollable events. 2016. [Online] https://hal.archives-ouvertes.fr/hal-01262444/.
17. F. B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3:30–50, 2000.
18. F. Somenzi. *CUDD: CU Decision Diagram Package*. ftp://vlsi.colorado.edu/pub/.
19. M. Wu, H. Zeng, and C. Wang. https://bitbucket.org/mengwu/shield-synthesis.
20. F. Yu, M. Alkhalaf, and T. Bultan. Patching vulnerabilities with sanitization synthesis. In *Intl. Conf. Software Engineering*, pages 251–260, 2011.