Synthesize Shield to Handle Burst Error

Meng Wu, Chao Wang, and Haibo Zeng

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

Abstract. Safety shield is an additional circuit or software attached to an reactive system to enforce its safety-critical properties when the system violates them. Existing methods for synthesizing safety shields for reactive systems can not tolerate burst error, but burst error is common in general failure modes of reactive system. In this paper, we extend existing methods to handle burst error. We have implemented our method in a tool and evaluated it on several sets of benchmarks. Our experiments show that our method is applicable for most of them and comparable with existing methods.

1 Introduction

In today's industry field, reactive systems are often safety-critical, and required to be validated or tested to guarantee its correctness. However, conventional validation and testing techniques on reactive systems are often disappointing under such requirements, due to scalability issues or IP rights concerns (e.g., third-party IP cores).

Fortunately, there are only a few properties in a reactive system that are safety-critical and for which we really want to ensure their correctness even though we believe they should be satisfied. In this setting, we would like to automatically construct a component, called the *shield*, that only enforce the correctness of the safety-critical properties and attach it to the design as illustrated in Fig. 1. The shield should meet two requirements: correctness which means it will corrects the erroneous outputs of the design, and minimum interference which means the shield will only change the outputs when it is necessary and the deviation should be kept as little as possible.

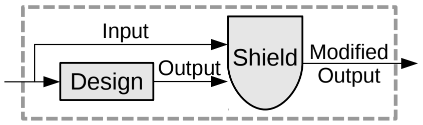
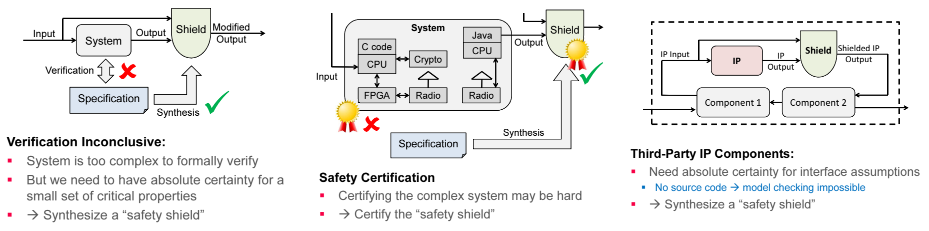


Fig. 1: Attaching a safety shield.

Besides, since the goal of *shield synthesis* is to track and regulate only a handful of critical properties, as opposed to the complete system specification, the shield synthesis is scalable and more practical. We argue the shield will be applicable to a wide variety of scenarios (Fig. 2) for designing safety-critical reactive applications. First, it can serve as remedy for inconclusive verification: when a critical property cannot be formally verified due to various reasons, we should synthesize a safety shield to enforce it at runtime. Second, it can speed up safety certification: instead of certifying the complex design itself, we should certify the much simpler shield, whose correctness guarantees the composed system is safe. Third, it can be used on third-party IP cores, whose source code is often unavailable, therefore making formal verification inapplicable.



Verification Inconclusive:
- System is too complex to formally verify
- But we need to have absolute certainty for a small set of critical properties
- → Synthesize a "safety shield"

Safety Certification
- Certifying the complex system may be hard
- → Certify the "safety shield"

Third-Party IP Components:
- Need absolute certainty for interface assumptions
  - No source code → model checking impossible
- → Synthesize a "safety shield"

Fig. 2: Applications: (1) remedy for verification; (2) simplify certification; and (3) shield 3rd-party IPs.

Existing methods for shield synthesis [3, 4] proposed a notion called k-stabilization to minimize the deviations and ensure the correctness at run time. A k-stabilizing shield takes at most k steps to correct a violation, and second violation is not allowed during the k steps. However, consecutive errors(called *burst error* in telecommunication) often happens when the design system fails due to environmental interference or external attacks. To handle such cases, we extended the existing k-stabilizing shield, so that it would always correct violations even for burst error. We also find some implications in the shield synthesis problem that can be used to simplify the synthesis problem.

The remainder of this paper is organized in the following manner. In Section 2, we formalized the problem in this paper and give some definition used in later sections. In Section 3, we introduce the idea to solve the safety game with help of some implications we found in this problem, and prove its validation. We then conduct a side to side comparison of k-stabilizing shield and our burst error tolerate shield in Section 4 with illustrative examples. We implement our algorithm in a tool and present experimental results on several sets of benchmarks in Section 5 and finally give our conclusions in Section 6.

2 Preliminaries

In this section, we establish the notations used in the rest of the paper.

Specifications. A *specification* φ defines a set $L(\varphi) \subseteq \Sigma^\omega$ of allowed traces. φ is *realizable* if there exists a design $\mathcal{D}$ that realizes it. $\mathcal{D}$ *realizes* φ, written $\mathcal{D} \models \varphi$, iff $L(\mathcal{D}) \subseteq L(\varphi)$. We assume that φ is a (potentially incomplete) set of *properties* $\{\varphi_1, \ldots, \varphi_l\}$ such that $L(\varphi) = \bigcap_i L(\varphi_i)$, and a design satisfies φ iff it satisfies all its properties. In this work, we are concerned with a *safety* specification φ^s, which is represented by an automaton $\varphi^s = (Q, q_0, \Sigma, \delta, F)$, where $\Sigma = \Sigma_I \cup \Sigma_O$, $\delta : Q \times \Sigma \to Q$, and $F \subseteq Q$ is a set of safe states. The *run* induced by trace $\overline{\sigma} = \sigma_0\sigma_1 \ldots \in \Sigma^\omega$ is the state sequence $\overline{q} = q_0 q_1 \ldots$ such that $q_{i+1} = \delta(q_i, \sigma_i)$. Trace $\overline{\sigma}$ (of a design $\mathcal{D}$) *satisfies* φ^s if the induced run visits only the safe states, i.e., $\forall i \geq 0 \,.\, q_i \in F$. The *language* $L(\varphi^s)$ is the set of all traces satisfying φ^s.

Safety Games. A (2-player, alternating) *game* is a tuple $\mathcal{G} = (G, g_0, \Sigma_I, \Sigma_O, \delta, F)$, where G is a finite set of game states, $g_0 \in G$ is the initial state, $F \subseteq G$ are the final states should not be visited, $\delta : G \times \Sigma_I \times \Sigma_O \to G$ is a complete transition function. The game is played by two players: the system and the environment. In every state $g \in G$ (starting with g_0), the environment first chooses an input letter $\sigma_I \in \Sigma_I$, and then the system chooses some output letter $\sigma_O \in \Sigma_O$. This defines the next state $g' = \delta(g, \sigma_I, \sigma_O)$, and so on. The resulting (infinite) sequence $\overline{g} = g_0 g_1 \ldots$ of game states is called a *play*. A play is *won* by the system iff $\forall i \geq 0 \,.\, g_i \in G \setminus F$

Synthesis Shield. Let $\mathcal{D} = (Q, q_0, \Sigma_I, \Sigma_O, \delta, \lambda)$ be a design, φ be a set of properties, and $\varphi^v \subseteq \varphi$ be a valid subset such that $\mathcal{D} \models \varphi^v$. Here, design should satisfy $\varphi^s = \varphi \setminus \varphi^v$, but it may not be guaranteed. So we synthesis another reactive system *shield* $\mathcal{S} = (Q', q'_0, \Sigma, \Sigma'_O, \delta', \lambda')$, which take both input and output of the design as input, and generate a corrected output which $(\mathcal{D} \circ \mathcal{S}) \models \varphi$ is guaranteed.

Composing Game Graph. During the shield synthesis problem, we need to compose two game together. If each game represent a property the shield need to satisfy, a composed game means the shield need to satisfy both properties at the same time. Hence the composition define as follows:

Compose $\mathcal{G}_1$ with $\mathcal{G}_2$ to $\mathcal{G} = (G, g_0, \Sigma = \Sigma_I \times \Sigma_O, \Sigma'_O, \delta, F)$, where

$G = G_1 \times G_2 = \{(q_1, q_2) \mid q_1 \in G_1, q_2 \in G_2\}$ is the state space of $\mathcal{G}$,

$g_0 = \{g_{01}, g_{02}\}$ is the new initial state,

$F = (F_1 \times G_2) \cup (G_1 \times F_2) = \{(q_1, q_2) \mid q_1 \in F_1$ or $q_2 \in F_2\}$ is the new error state space. in another word, a state $q = (q_1, q_2)$ is safe if and only if q_1 is safe in $\mathcal{G}_1$ and q_2 is also safe in $\mathcal{G}_2$,

$\delta : \{G \times \Sigma \times \Sigma'_O \rightarrow G' \mid (q_1, q_2) \in G, (q'_1, q'_2) \in G', q_1 \times \Sigma \times \Sigma'_O \rightarrow q'_1 \in \delta_1, q_2 \times \Sigma \times \Sigma'_O \rightarrow q'_2 \in \delta_2\}$

Winning Region Computation. We use classical winning region computation algorithm [8] to solve safety games. In this algorithm, we compute "attractors" for a subset of safe states and final states, until reach a fix-point.

More specifically, for a game $\mathcal{G}$, start with safe states $S = G \setminus F$ and final states F. After one iteration of computation, $S_{new} = S \setminus Q$ and $F_{new} = F \cup Q$, where $Q = \{q \mid \exists \sigma_I \in \Sigma . \forall \sigma_O \in \Sigma'_O . q' = \delta(q, \sigma_I, \sigma_O) \wedge (q' \in F)\}$. After that, start another iteration with safe states as S_{new} and final states as F_{new} until reach a fix-point on safe states and final states.

Winning Region Abstraction on Composed Game. Given $\mathcal{G} = \mathcal{G}_1 \times \mathcal{G}_2$, with final states $F = \{(q_1, q_2) \mid q_1 \in F_1$ or $q_2 \in F_2\}$, after one iteration of winning states computation, assume $F_{1new} = F_1 \cup Q_1$, define set of states $K \subseteq G$ that $K = \{(q_1, q_2) \mid q_1 \in Q_1, q_2 \in G_2\}$, so $\forall k \in K . \exists \sigma_I \in \Sigma . \forall \sigma_O \in \Sigma'_O . k' = \delta(k, \sigma_I, \sigma_O) \wedge (k' \in F)$

This is because any state in the game is left complete, which means for any input and output alphabet, it has corresponding outgoing edge. Suppose $q_1 \in Q_1$, that for specific input alphabet σ_I, all its outgoing edges leads to final states in F_1. Then for any $q_2 \in G_2$, there is state $q = (q_1, q_2) \in G$, that for any the same specific input alphabet σ_I, all its outgoing edges leads to states $q' = (q'_1, q'_2)$ where $q'_1 \in F_1$. As we defined final states in $\mathcal{G}$ above, any state in $\mathcal{G}$ with its subnode in Q_1 is also a new final state in $\mathcal{G}$.

Intuitively, if we compute winning region simultaneously on $\mathcal{G}_1$ and $\mathcal{G}$, after each iteration, we will have F_1 be the current final states for $\mathcal{G}_1$, F be the current final states in $\mathcal{G}$, and $\{(q_1, q_2) \mid q_1 \in F_1, q_2 \in G_2\} \subseteq F$. Additionally, $\mathcal{G}$ will reach fix-point no earlier than $\mathcal{G}_1$. Therefore, when both $\mathcal{G}_1$ and $\mathcal{G}$ reach fix-point, we have winning region of $\mathcal{G}_1$ be $\mathcal{W}_1 = G_1 \setminus F_1$, and $\mathcal{W} = (G \setminus F) \subseteq \{(q_1, q_2) \mid q_1 \in \mathcal{W}_1, q_2 \in \mathcal{W}_2\}$

In such a case, if we want to compute winning region of a composed game $\mathcal{G}$, one alternative way is to compute winning region of $\mathcal{G}_1$ and $\mathcal{G}_2$ separately, and then using the composed winning region $\mathcal{W}_1 \times \mathcal{W}_2$ as a starting point to compute winning region of $\mathcal{G}$, since we know $\mathcal{W}_1 \times \mathcal{W}_2$ is an over-approximation of $\mathcal{W}$.

3 Game Solving using Implications

In this section, we will first briefly explained the shield synthesis procedure, and then we will analyze the implication behind shield synthesis problem, and show how to use it to simplify the synthesis problem.

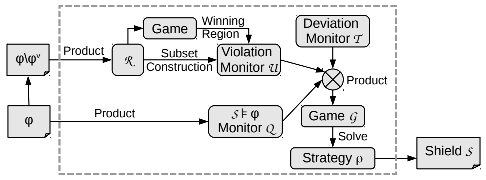


Fig. 3: Outline of shield synthesis procedure.

In our algorithm, we still follow the generic procedure as Fig. 3. The final game graph is composed by three automata: violation monitor $\mathcal{U}$, deviation monitor $\mathcal{T}$, correctness monitor $\mathcal{Q}$. Here, violation monitor $\mathcal{U} = (U, u_0, \Sigma, \delta^u)$ is an automaton with only input alphabet, and each state is marked with a counter $c \in \{0, \ldots, k\}$. Let $U_1 \in U$ be the states with $c = 0$, and $U_2 \in U$ be the states with $c > 0$. It is an automaton to monitor whether the design has violate the property or not, and also a guessing automaton to track the possible correct behavior of the design. The deviation monitor $\mathcal{T} = (T, t_0, \Sigma_O \times \Sigma_O', \delta^t)$, where $T = \{t_0, t_1\}$ and $\delta^t(t, (\sigma_O, \sigma_O{}')) = t_0$ iff $\sigma_O = \sigma_O{}'$. It is build to specify whether the output of the shield and design is identical. And the correctness monitor $\mathcal{Q}$ is an automaton with respect to Σ and Σ_O', to ensure $(\mathcal{D} \circ \mathcal{S}) \models \varphi$.

As we mentioned in Section 1, the shield should meet two requirements: (1) minimum interference: the shield should change the output only when its necessary; (2) correctness: the output of the shield should ensure the specification to be satisfied. According to this, we have the synthesis procedure into two parts: (1) $\mathcal{G}_1 = \mathcal{U} \times \mathcal{T}$ for minimum interference and (2) $\mathcal{G}_2 = \mathcal{Q}$ for correctness.

In our algorithm, the error state $F_1 \in G_1$ is defined as $F_1 = \{(u, t) \mid c_u = 0, t = t_1\}$, that is $\mathcal{G}_1$ specify the shield should not change the output if the design dose not violate the property. We represent winning region of $\mathcal{G}_1$ as $\mathcal{W}_1$, then $\mathcal{W}_1 = G_1 \setminus F_1$, that is exactly the non-error states. It can be make out that any state in $\mathcal{G}_1$ is always free to choose output alphabet(just let $\sigma_O = \sigma_O{}'$) to avoid going to error states.

With correctness automaton represented by $\mathcal{G}_2$, the $\mathcal{G} = \mathcal{G}_1 \times \mathcal{G}_2$ is the final game the existing method use to solve the safety game on. As we proved in Section 2, $\mathcal{W} \subseteq \mathcal{W}_1 \times \mathcal{W}_2$. But why $\mathcal{W} \neq \mathcal{W}_1 \times \mathcal{W}_2$?

In general case, assume one state $\{q_1 \in G_1 \mid \exists \sigma_I \in \Sigma.(\exists \sigma_O \in \Sigma_O'.\delta(q_1, \sigma_I, \sigma_O) \in F_1) \wedge (\exists \sigma_O \in \Sigma_O'.\delta(q_1, \sigma_I, \sigma_O) \notin F_1)\}$, that is for certain input alphabet σ_I, q_1 can goes to either final states or not, so q_1 should be considered as a "safe" state in $\mathcal{G}_1$. If $q_2 \in G_2$ is also the case: for the same input alphabet, it could go to either a final state or not. The only difference is for the same output alphabet q_1 will go to a final state, but

q_2 will not, and the vice verse. Therefore, for $q = (q_1.q_2) \in G$, given σ_I will always lead q to a final state, and make q not a "safe" state in $\mathcal{G}$. That's why $\mathcal{W}$ is only an under-approximation of $\mathcal{W}_1 \times \mathcal{W}_2$

However, we found this would not happen in our synthesis procedure, because both violation monitor and correctness monitor are derived from specification automaton. They share the same structure for states with violation counter equal to 0, but different alphabet label. Hence there is an implication behind this: if a certain input alphabet $\sigma_I \times \sigma_O$ lead the design to violate property, where $c > 0$, at the same time if ${\sigma_O}' = \sigma_O$, the correctness monitor will also goes to final states. In another word, if the violation monitor stay in states with $c = 0$ and ${\sigma_O}' = \sigma_O$, the shield will never violate correctness monitor as well, that is if the design hold the property, and the shield maintain the same output, the whole system should valid. We will prove it formally by contradiction.

Let us assume $\mathcal{W} \cup Q = \mathcal{W}_1 \times \mathcal{W}_2$, where $Q = \{q \mid \exists \sigma_I \in \Sigma . \forall \sigma_O \in \Sigma'_O . \delta(q, \sigma_I, \sigma_O) = q' \in F\}$, and also for $(q_1, q_2) \in Q, q_1 \in \mathcal{W}_1$ and $q_2 \in \mathcal{W}_2$

As we explained above, there exist an input alphabet $\sigma_{Ie} \times \sigma_{Oe} \in \Sigma$, all possible $\sigma_O \in \Sigma'_O$ will lead state $q = (q_1, q_2)$ to a final state:

Contradiction 1: if ${\sigma_O}'_e = \sigma_{Oe}$ and the violation counter of $q'_1 = \delta_1(q_1, \sigma_{Ie} \times \sigma_{Oe}, {\sigma_O}'_e)$ is 0. Then q'_1 cannot be a final state according to the definition of F_1. Since $q' = (q'_1, q'_2)$ is a final state, $q'_2 = \delta_2(q_2, \sigma_{Ie} \times \sigma_{Oe}, {\sigma_O}'_e)$ must be a final state in correctness monitor. Which is contradictory to the implication we found.

Contradiction 2: if ${\sigma_O}'_e \neq \sigma_{Oe}$, $q'_1 = \delta_1(q_1, \sigma_{Ie} \times \sigma_{Oe}, {\sigma_O}'_e)$ with a counter $c = 0$ will not be a final state according to the definition of F_1. In a similar way, with the same input and output alphabet, the next state of q_2 must be a final state in correctness monitor. It is also contradictory to our implication mentioned above.

In a word, there does not exist such a set of Q such that $\mathcal{W} \cup Q = \mathcal{W}_1 \times \mathcal{W}_2$, which means$\mathcal{W} = \mathcal{W}_1 \times \mathcal{W}_2$! When considering this implication into the shield synthesis problem, we can greatly simplify the synthesis procedure to three steps: (1) compute $\mathcal{W}_1$ as the non-error states of $\mathcal{G}_1$; (2) solving the game on $\mathcal{G}_2$ to get $\mathcal{W}_2$; (3) just compose $\mathcal{W}_1$ with $\mathcal{W}_2$ to get the final winning region, from which we can extract a wining strategy for the shield.

4 Burst Error Tolerate Shield V.S. *k*-stabilizing Shield

Our shield is aimed to correct the violations of critical property of the design. Such violations of design usually happen due to environmental interference, eg. radiation, RAM failure, and they usually cause several consecutive violations in a very short time.

In [3, 4] paper, they defined a concept of k-stabilizing shield: When a property violation by the design $\mathcal{D}$ becomes unavoidable, the shield $\mathcal{S}$ is allowed to deviate from the designs outputs for at most k consecutive time steps, including the current step. Only after these k steps, the next violation is tolerated. Therefore, when k = 1, it is a perfect shield that could fix an error immediately. While when k=2, the shield need 2 steps to completely fix an error, which means if the design produces 2 errors consecutively, the shield will give up monitoring the design anymore.

What we propose, is a shield that keep correcting consecutive violations without giving up. This is done by modifying the error tracking automaton which is responsible for motoring the behavior of design: In k-stabilizing shield synthesis, the error tracking automaton optimistically assume the design will not make another mistake, so it will only start guessing the design‘s behavior when it have to make mistakes; while in our synthesis, we conservatively assume the design will make mistake at any time, so whenever there is a chance for the design to make mistake, we generate a abstract state to guessing its correct behaviors.

Example: We take a simple property of ARM bus arbiter from [2] as example. The property could be represented by LTL formula: $\mathsf{G}(\neg R \rightarrow \mathsf{X}(\neg S))$, which means the transmission cannot be started (by symbol S) if the it is not ready (by symbol R). When translated into automaton, it is shown as Fig. 4. To construct an error tracking automaton, we first need to replace error state $S2$ with a guessing state, that is the right most state labeled with $S0'S1'$ in Fig. **??**, it means the system may goes to either $S0$ or $S1$ instead of violation. Starting from it, we keep guessing the behavior of the system by adding another two guessing states(dashed states in Fig. **??**). Both dashed states contain error state $S2$, so they will be again replaced with guessing states. Fig. **??** shows the final error tracking automaton, $S2'$ are replaced with another guessing state $S0'$ and $S1'$ respectively. Since $S0'$ and $S1'$ are guessing states, they cannot be merged back to normal states. While in K-stabilizing shield synthesis, the guessing error state $S2$ will just be removed from dashed states: so only one state will be left in the two guessing state, which is shown as in Fig. **??**. Finally, the resulting error tracking automaton is shown as Fig. **??**, that two guessing states are merged back to normal state after removing $S2'$.

Synthesis based on k=1 error tracking automaton as Fig. **??** will failed on solving the safety problem, since one step is not enough to completely recover the violation, or in another word, one step is not enough to guess and confirm the actual behavior of the design. For the 1-stabilizing error tracking automaton as Fig. **??**, if the design output another $\neg R \wedge S$ when the shield is in state $S0'S1'$, it has to face a dilemma: the shield should not change the output according to minimum interference requirement, because $\neg R \wedge S$ leads to a non-violation state $S1$; In another way, if the design‘s actual state is $S1$, which is possible because the guessing state is either $S0'$ or $S1'$, the correctness requirement will be undermined if the shied also outputs $\neg R \wedge S$. Our error tracking automaton will keep guessing any possible behaviors of the design while encountering possible violations, it will alway generate a shield acting like 1-stabilizing shield.

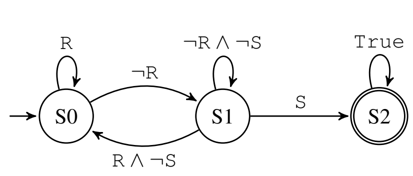
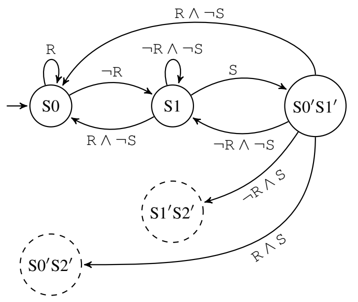
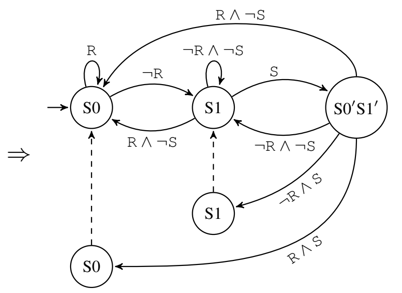


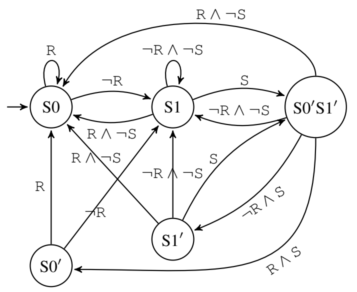
Fig. 4: Specification of AMBA example.



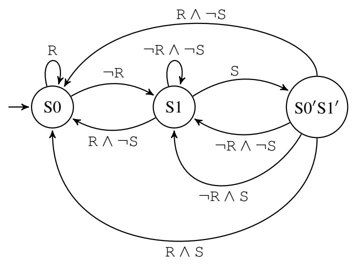
(a)Intermediate Error Tracking Automaton



(b)Intermediate Error Tracking Automaton



(c)ET DFA of Burst Error Shield



(d)ET DFA of 1-stabilizing Shield

Fig. 5: Construction of Error Tracking Automaton.

5 Experimental Results

We evaluate our method on safety specifications from various sources, including (1) Toyota powertrain control verification benchmark [7], (2) properties for engine and brake controls [9], (3) traffic light controller from VIS [5], (4) selected properties for ARM AMBA bus arbiter [2], and (5) LTL (Linear Temporal Logic) property patterns from Dwyer et al. [6]. More details about the benchmarks can be found in the supplementary documents in our tool repository [1]. All experiments are performed on a machine with an Intel i5 3.1GHz CPU and 4GB RAM.

Table 1 shows the evaluation results for benchmarks (1)–(4). As input, we include the name and size (number of states and edges) of specification. We compared the output of three algorithms: the existing K-stability algorithm, our standard algorithm handling burst errors, and our extensive algorithm using implications in winning strategy computation. For each algorithm, we list the output as the guard size(number of states), and the synthesis time in seconds. We also give the K value in K-stability algorithm. The properties from Toyota powertrain control verification benchmark are on the model of a fuel control system [7], 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 from engine and brake controls include Req 1.1–Req 3 in our illustrative example. The properties for traffic light control [5] are for safety of a crossroads traffic light. The AMBA benchmark [2] includes combinations of various properties of an

Table 1: Evaluation results for Benchmarks (1)–(4).

Property φ	S/E in φ	K-Stability			Burst Error		Implication	
		k	$\lvert\mathcal{D}'\rvert$	Time	$\lvert\mathcal{D}'\rvert$	Time	$\lvert\mathcal{D}'\rvert$	Time
Toyota powertrain [7]	23/241	1	778	0.27	778	0.35	1188	0.36
Engine and brake ctrl [9]	5/93	1	20	0.1	20	0.11	40	0.08
Traffic light [5]	4/97	1	13	0.09	27	0.2	39	0.2
AMBA G1+2+3 [2]	12/131	1	186	0.11	186	0.11	363	0.13
AMBA G1+2+4 [2]	8/197	2	150	6.31	193	2.31	280	2.27
AMBA G1+3+4 [2]	15/245	2	933	55.61	3868	99.52	5628	97.63
AMBA G1+2+3+5 [2]	18/296	2	1609	191.89	7183	58.43	12240	61.8
AMBA G1+2+4+5 [2]	12/621	2	244	3992.9	1025	479.2	1276	472.9
AMBA G4+5+6 [2]	26/865	3	1599	117.93	1827	26.31	3475	26.4
AMBA G5+6+10 [2]	31/1261	3	2221	9.87	3135	30.5	9780	31.4
AMBA G5+6+9e4+10 [2]	50/2005	3	4582	17.64	8270	40.43	19992	42.1
AMBA G5+6+9e8+10 [2]	68/2713	3	9555	34.94	20466	80.8	52260	86.8
AMBA G5+6+9e16+10 [2]	104/4129	3	24505	74.75	55786	194.2	144818	189.7
AMBA G5+6+9e64+10 [2]	320/12625	3	254317	1080.8	637098	2865.62	1733446	2182.5
AMBA G8+9e4+10 [2]	48/1745	3	3860	7.06	5550	5.61	8836	6.13
AMBA G8+9e8+10 [2]	84/3197	3	13120	22.54	36284	30.53	50381	33.7
AMBA G8+9e16+10 [2]	156/6101	3	47864	83.77	128924	111.07	183915	103.1
AMBA G8+9e64+10 [2]	588/23523	3	710600	2274.2	2734572	7843	3037725	2271.5

ARM bus arbiter. For example, property G2 is in the form $\mathsf{G}(q \rightarrow \mathsf{X}(r\,\mathsf{W}\,p))$, where G means globally, X means *Next*, and W is the *Weakuntil* operator in LTL.

To further demonstrate the feasibility of our method, we synthesize guards for some of the most popular LTL property patterns in verification [6]. The evaluation results are shown in Table 2. Column 1 shows the LTL formula of each property, which contains temporal operators such as *Finally* (i.e., $\mathsf{F}\,p$, meaning p must be true eventually), as well as more complex operators like *Until* (i.e., $p\,\mathsf{U}\,r$, meaning p has to be true at least until r) or *Weakuntil* (i.e., $p\,\mathsf{W}\,r \equiv \mathsf{G}\,p \vee (p\,\mathsf{U}\,r)$). Since some of the properties contain *liveness* aspects, we bound the number of reaction time steps (b) to obtain the desired safety specification.

The results show that, for most benchmarks, our method is comparable with existing K-stability algorithm. The implication is effective for some benchmarks like $\mathsf{F}\,p$ and $\mathsf{G}(\neg q) \vee \mathsf{F}(q \wedge \mathsf{F}\,p)$ with large bound values in Table 2. For some other properties that implication does not make big difference, because the construction of error tracking automaton takes most of synthesis time, and the implication only helps on the winning strategy computation. Anyway, implication wound not make it much worse in all cases. For a small part of properties(e.g. the last several rows in Table 2), our methods take significant longer time than K-stability algorithm.

Another notation need to be pointed out is, most benchmarks in Table 1 result in shields with $k > 1$ which cannot handle burst errors, but our methods are guaranteed to synthesize a shield that resistants to burst errors.Furthermore, our experiments on LTL property patterns indicate the applicability to a broad range of applications with similar forms of properties.

Table 2: Evaluation results for Benchmark (5).

Property φ	b	S/E in φ	K-Stability			Burst Error		Implication	
			k	$\lvert\mathcal{D}'\rvert$	Time	$\lvert\mathcal{D}'\rvert$	Time	$\lvert\mathcal{D}'\rvert$	Time
$\mathsf{G}\,\neg p$	-	2/3	1	3	0.01	3	0.01	3	0.01
$\mathsf{F}\,r \rightarrow (\neg p\,\mathsf{U}\,r)$	-	4/7	1	12	0.01	12	0.01	15	0.01
$\mathsf{G}(q \rightarrow \mathsf{G}(\neg p))$	-	3/5	1	7	0.01	7	0.01	8	0.01
$\mathsf{G}((q \wedge \neg r \wedge \mathsf{F}\,r) \rightarrow (\neg p\,\mathsf{U}\,r))$	-	4/10	1	12	0.01	12	0.01	15	0.05
$\mathsf{G}(q \wedge \neg r \rightarrow (\neg p\,\mathsf{W}\,r))$	-	3/8	1	6	0.01	9	0.01	10	0.04
$\neg r\,\mathsf{W}(p \wedge \neg r)$	-	3/5	1	6	0.01	9	0.01	10	0.01
$\mathsf{G}(q \wedge \neg r \rightarrow (\neg r\,\mathsf{W}(p \wedge \neg r)))$	-	3/8	1	6	0.01	11	0.01	14	0.01
$\mathsf{F}\,p$	0	3/4	1	5	0.01	5	0.01	8	0.01
$\mathsf{F}\,p$	4	7/12	1	23	0.01	23	0.01	48	0.01
$\mathsf{F}\,p$	16	19/36	1	173	0.06	173	0.06	360	0.06
$\mathsf{F}\,p$	64	67/132	1	2213	0.76	2213	0.77	4488	0.56
$\mathsf{F}\,p$	256	259/516	1	33413	46.93	33413	45.52	67080	10.53
$\mathsf{F}\,p$	512	515/1028	1	132357	509.1	132357	511.08	265224	54.46
$\mathsf{G}(\neg q) \vee \mathsf{F}(q \wedge \mathsf{F}\,p)$	0	3/5	1	6	0.01	9	0.01	10	0.01
$\mathsf{G}(\neg q) \vee \mathsf{F}(q \wedge \mathsf{F}\,p)$	4	7/13	1	19	0.02	19	0.01	48	0.01
$\mathsf{G}(\neg q) \vee \mathsf{F}(q \wedge \mathsf{F}\,p)$	16	19/37	1	157	0.06	157	0.06	360	0.05
$\mathsf{G}(\neg q) \vee \mathsf{F}(q \wedge \mathsf{F}\,p)$	64	67/133	1	2149	0.79	2149	0.8	4488	0.62
$\mathsf{G}(\neg q) \vee \mathsf{F}(q \wedge \mathsf{F}\,p)$	256	259/517	1	33157	46.91	33157	46.27	67080	10.78
$\mathsf{G}(\neg q) \vee \mathsf{F}(q \wedge \mathsf{F}\,p)$	512	515/1029	1	131845	517.79	131845	668.1	265224	54.55
$\mathsf{G}(q \wedge \neg r \rightarrow (\neg r\,\mathsf{U}(p \wedge \neg r)))$	2	4/11	1	10	0.01	24	0.01	33	0.01
$\mathsf{G}(q \wedge \neg r \rightarrow (\neg r\,\mathsf{U}(p \wedge \neg r)))$	4	6/17	1	34	0.02	216	0.08	265	0.08
$\mathsf{G}(q \wedge \neg r \rightarrow (\neg r\,\mathsf{U}(p \wedge \neg r)))$	8	10/29	1	276	0.26	7152	3.92	8037	4.54
$\mathsf{G}(q \wedge \neg r \rightarrow (\neg r\,\mathsf{U}(p \wedge \neg r)))$	10	12/35	1	740	1.17	35820	63.54	39391	60.1
$\mathsf{G}(q \wedge \neg r \rightarrow (\neg r\,\mathsf{U}(p \wedge \neg r)))$	12	14/41	1	1958	6.34	172008	1506.9	186329	1414.5
$\mathsf{G}(q \wedge \neg r \rightarrow (\neg r\,\mathsf{U}(p \wedge \neg r)))$	14	16/47	1	5150	42.23	-	TO	-	TO

6 Conclusions

We have proposed the notation of burst error tolerate shield, and a new method to simplify the game solving process during the shield synthesis by using the implications we introduced in Section 3. We have implemented our method based on the work of [3, 4]. Our results show that our method successfully generated shields that could handle burst error for almost all benchmarks, while the existing method cannot produce 1-stabilizing shields for some of them. Besides, our method has demonstrate a comparable or even better performance as the existing method. The implication extension is also proved to be effective on some particular benchmarks.

References

1. https://bitbucket.org/mengwu/shield-synthesis.
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 *International Conference on Tools and Algorithms for Construction and Analysis of Systems*. Springer, 2015.
4. R. Bloem, B. Könighofer, R. Könighofer, and C. Wang. Shield synthesis: Runtime enforcement for reactive systems. *CoRR*, abs/1501.02573, 2015.
5. R. K. Brayton et al. VIS: A system for verification and synthesis. In *CAV*, LNCS 1102, pages 428–432. Springer, 1996.
6. M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *ICSE*, pages 411–420. ACM, 1999.
7. Xiaoqing Jin, Jyotirmoy V. Deshmukh, James Kapinski, Koichi Ueda, and Ken Butts. Powertrain control verification benchmark. In *HSCC*, 2014.
8. R. Mazala. Infinite games. In *Automata, Logics, and Infinite Games: A Guide to Current Research*, LNCS 2500, pages 23–42. Springer, 2001.
9. NHTSA. *49 CFR Part 571: Federal Motor Vehicle Safety Standards; Accelerator Control Systems*. Department of Transportation, 2012.