Form Methods Syst Des (2017) 51:332–361
DOI 10.1007/s10703-017-0276-9

CrossMark

Shield synthesis

Bettina Könighofer[1] · Mohammed Alshiekh[2] · Roderick Bloem[1] · Laura Humphrey[3] · Robert Könighofer[1] · Ufuk Topcu[2] · Chao Wang[4]

Published online: 25 September 2017
© The Author(s) 2017. This article is an open access publication

Abstract *Shield synthesis* is an approach to enforce safety properties at runtime. A shield monitors the system and corrects any erroneous output values instantaneously. The shield deviates from the given outputs as little as it can and recovers to hand back control to the system as soon as possible. In the first part of this paper, we consider shield synthesis for reactive hardware systems. First, we define a general framework for solving the shield synthesis problem. Second, we discuss two concrete shield synthesis methods that automatically construct shields from a set of *safety* properties: (1) *k-stabilizing* shields, which guarantee

Supported by the Austrian Science Fund (FWF) through the projects RiSE (S11406-N23) and LogiCS (W1255-N23), the European Commission through the project IMMORTAL (644905), and the Air Force Research Laboratory Office of Scientific Research (AFOSR) (17RQCOR417).

✉ Bettina Könighofer
Bettina.Konighofer@iaik.tugraz.at

Mohammed Alshiekh
malshiekh@utexas.edu

Roderick Bloem
Roderick.Bloem@iaik.tugraz.at

Laura Humphrey
laura.humphrey@us.af.mil

Robert Könighofer
Robert.Konighofer@iaik.tugraz.at

Ufuk Topcu
utopcu@utexas.edu

Chao Wang
wang626@usc.edu

1 IAIK, Graz University of Technology, Graz, Austria

2 University of Texas at Austin, Austin, TX, USA

3 Control Science Center of Excellence, AFRL, Wright-Patterson AFB, Fairborn, OH, USA

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

recovery in a finite time. (2) *Admissible* shields, which attempt to work with the system to recover as soon as possible. Next, we discuss an extension of k-stabilizing and admissible shields, where erroneous output values of the reactive system are corrected while liveness properties of the system are preserved. Finally, we give experimental results for both synthesis methods. In the second part of the paper, we consider shielding a human operator instead of shielding a reactive system: the outputs to be corrected are not initiated by a system but by a human operator who works with an autonomous system. The challenge here lies in giving simple and intuitive explanations to the human for any interferences of the shield. We present results involving mission planning for unmanned aerial vehicles.

Keywords Synthesis · Runtime reinforcement · Games · Human factors · UAV

1 Introduction

Technological advances enable the development of increasingly sophisticated systems. Smaller and faster microprocessors, wireless networking, and new theoretical results in areas such as machine learning and intelligent control are paving the way for transformative technologies across a variety of domains—self-driving cars that have the potential to reduce accidents, traffic, and pollution; and unmanned systems that can safely and efficiently operate on land, under water, in the air, and in space. However, in each of these domains, concerns about safety are being raised [13,27]. Specifically, there is a concern that due to the complexity of such systems, traditional test and evaluation approaches will not be sufficient for finding errors, and alternative approaches such as those provided by formal methods are needed [28].

Formal methods are often used to verify systems at design time, but this is not always realistic. Some systems are simply too large to be fully verified. Others, especially systems that operate in rich dynamic environments or those that continuously adapt their behavior through methods such as machine learning, cannot be fully modeled at design time. Still others may incorporate components that have not been previously verified and cannot be modeled, e.g., pre-compiled code libraries. Also, even systems that have been fully verified at design time may be subject to external faults such as those introduced by unexpected hardware failures or human inputs. One way to address this issue is to model nondeterministic behaviours (such as faults) as disturbances, and to verify the system with respect to this disturbance model [29]. However, it may be impossible to model all potential unexpected behavior at design time.

An alternative in such cases is to perform *runtime verification* to detect violations of specified properties while a system is executing [24]. An extension of this idea is to perform *runtime enforcement* of specified properties, in which violations are not only detected but also overwritten in such a way that specified properties are maintained.

In this paper, we discuss a general approach for runtime enforcement called *shield synthesis*. From the specified properties, we automatically construct a component, called the shield, that monitors the input/output of the system and instantaneously overwrites incorrect outputs as illustrated in Fig. 1.

A shield must ensure both *correctness*, i.e., it corrects system outputs such that all properties are always satisfied, as well as *minimum deviation*, i.e., it deviates from system outputs only if necessary and as rarely as possible. The latter requirement is important because the system may satisfy additional noncritical properties that are not considered by the shield but should be retained as much as possible. Shield synthesis is a promising new direction

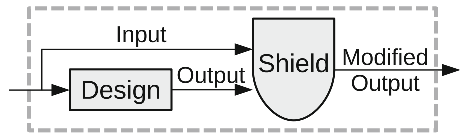


Fig. 1 Attaching a safety shield

for synthesis in general, because it uses the strengths of reactive synthesis while avoiding its weaknesses—the set of critical properties can be small and relatively easy to specify regardless of the implementation details of a complex system—which implies scalability and usability.

In the first part of this paper, we consider shield synthesis for reactive hardware systems. Here, we first define a general framework for solving the shield synthesis problem. Second, we discuss two concrete shield synthesis methods to automatically construct a shield from a set of safety properties. The resulting shields are called k-stabilizing shields and admissible shields.

k-stabilizing shields guarantee recovery in a finite time. Since we are given a safety specification, we can identify wrong outputs, that is, outputs after which the specification is violated (more precisely, after which the environment can force the specification to be violated). A wrong trace is then a trace that ends in a wrong output. k-stabilizing shields modify the outputs so that the specification always holds, but that such deviations last for at most k consecutive steps after a wrong output.

Admissible shields overcome the following shortcoming of k-stabilizing shields: The k-stabilizing shield synthesis problem is unrealizable for many safety-critical systems, because a finite number of deviations cannot be guaranteed. To address this issue, admissible shields guarantee the following: (1) For any wrong trace, if there is a finite number k of steps within which the recovery phase can be guaranteed to end, an admissible shield takes an *adversarial* view on the system and will always achieve this. Admissible shields are subgame optimal and guarantee to end the recovery phase for any state for the smallest k possible if such a k exists for that state. (2) The shield is *admissible* in the following sense: for any state in which there is no such number k, it takes a *collaborative* view on the system and always picks a deviation that is optimal in that it ends the recovery phase as soon as possible for some possible future inputs. As a result, admissible shields work well in settings in which finite recovery cannot be guaranteed, because they guarantee correctness and may well end the recovery period if the system does not pick adversarial outputs.

k-stabilizing shields and admissible shields enforce critical safety properties and ensure minimum deviation, such that other noncritical properties of the system that are not considered by the shield are retained as much as possible. In addition to critical safety properties, many systems must also meet critical liveness properties. However, a challenge for enforcing liveness properties using shields is that liveness property violations cannot be detected at any finite point in time (at any point, the property may still be satisfied in the future). Due to the minimum deviation property of shields, a shield would have to delay enforcing a liveness property as long as possible, and since liveness properties can always be satisfied at some point in the future, the shield in practice would never enforce the liveness property. So rather than enforcing liveness properties, we focus on retaining liveness properties under the assumption that the shielded system satisfies them. We therefore conclude the first part of this paper with an extension of the k-stabilizing and the admissible shield synthesis procedure that allows liveness-preserving corrections of the system's output.

In the second part of the paper, we consider shielding a human operator who works with an autonomous system instead of shielding a reactive system: the outputs to be corrected are not

initiated by a system but by a human operator. When shielding human operators we attach the shield *before* the operator. We call this type of shield a *preemptive shield*. The shield acts each time the operator is to make a decision and provides a list of safe outputs. This list restricts the choices for the operator. Additionally, when shielding a human operator, it is necessary to provide simple and intuitive explanations to the operator for any interferences of the shield. We call shields able to provide such explanations *explanatory shields*. We motivate the need for shielding a human operator via a case study involving mission planning for an unmanned aerial vehicle (UAV).

1.1 Outline

The remainder of this paper is organized as follows. First, we establish notation in Sect. 2. In Sect. 3 we discuss shield synthesis for reactive hardware systems. We begin this section by using an example to illustrate the technical challenges and our solution approach in Sect. 3.1. We formalize the problem in a general framework for shield synthesis in Sect. 3.2. In Sects. 3.3, 3.4, 3.5, and 3.6 we define and describe the synthesis procedure for k-stabilizing shields and for admissible shields. Section 3.7 describes an alternative construction for k-stabilizing and admissible shields, and Sect. 3.8 discusses liveness-preserving shielding. To conclude the first part of the paper, we provide experimental results for both shield synthesis approaches in Sect. 3.9. In Sect. 4 we consider shielding a human operator instead of shielding a reactive system. In this setting, we discuss preemptive shields in Sect. 4.1 and explanatory shields in Sect. 4.2. We conclude the second part of the paper with a case study on UAV mission planning in Sect. 4.3 Finally, we give an overview on related work in Sect. 5 and conclude in Sect. 6.

2 Preliminaries

We denote the Boolean domain by $\mathbb{B} = \{\top, \bot\}$, the set of natural numbers by $\mathbb{N}$, and abbreviate $\mathbb{N} \cup \{\infty\}$ by $\mathbb{N}^\infty$. The set of finite (infinite) words over an alphabet Σ is denoted by Σ^* (Σ^ω), and $\Sigma^\infty = \Sigma^* \cup \Sigma^\omega$. We will also refer to words as *(execution) traces*. We write $|\overline{\sigma}|$ for the length of a trace $\overline{\sigma} \in \Sigma^*$. A set $L \subseteq \Sigma^\infty$ of words is called a *language*. We denote the set of all languages as $\mathcal{L} = 2^{\Sigma^\infty}$. We consider a finite set $I = \{i_1, \ldots, i_m\}$ of Boolean inputs and a finite set $O = \{o_1, \ldots, o_n\}$ of Boolean outputs. The input alphabet is $\Sigma_I = 2^I$, the output alphabet is $\Sigma_O = 2^O$, and $\Sigma = \Sigma_I \times \Sigma_O$. For $\overline{\sigma_I} = x_0 x_1 \ldots \in \Sigma_I^\infty$ and $\overline{\sigma_O} = y_0 y_1 \ldots \in \Sigma_O^\infty$, we write $\overline{\sigma_I} || \overline{\sigma_O}$ for the composition $(x_0, y_0)(x_1, y_1) \ldots \in \Sigma^\infty$.

2.1 Reactive systems

A *Mealy machine* (a reactive system, also called a design) is a 6-tuple $\mathcal{D} = (Q, q_0, \Sigma_I, \Sigma_O, \delta, \lambda)$, where Q is a finite set of states, $q_0 \in Q$ is the initial state, $\delta : Q \times \Sigma_I \to Q$ is a complete transition function, and $\lambda : Q \times \Sigma_I \to \Sigma_O$ is a complete output function. Given the input trace $\overline{\sigma_I} = x_0 x_1 \ldots \in \Sigma_I^\infty$, the system $\mathcal{D}$ produces the output trace $\overline{\sigma_O} = \mathcal{D}(\overline{\sigma_I}) = \lambda(q_0, x_0)\lambda(q_1, x_1) \ldots \in \Sigma_O^\infty$, where $q_{i+1} = \delta(q_i, x_i)$ for all $i \geq 0$. The set of words produced by $\mathcal{D}$ is denoted $L(\mathcal{D}) = \{\overline{\sigma_I} || \overline{\sigma_O} \in \Sigma^\infty \mid \mathcal{D}(\overline{\sigma_I}) = \overline{\sigma_O}\}$.

Let $\mathcal{D} = (Q, q_0, \Sigma_I, \Sigma_O, \delta, \lambda)$ and $\mathcal{D}' = (Q', q_0', \Sigma, \Sigma_O, \delta', \lambda')$ be two reactive systems. A serial composition of $\mathcal{D}$ and $\mathcal{D}'$ is realized if the input and output of $\mathcal{D}$ are fed to $\mathcal{D}'$. We denote such composition as $\mathcal{D} \circ \mathcal{D}' = (\hat{Q}, \hat{q_0}, \Sigma_I, \Sigma_O, \hat{\delta}, \hat{\lambda})$, where $\hat{Q} = Q \times Q'$,

$\hat{q}_0 = (q_0, q_0')$, $\hat{\delta}((q, q'), \sigma_I) = (\delta(q, \sigma_I), \delta'(q', (\sigma_I, \lambda(q, \sigma_I))))$, and $\hat{\lambda}((q, q'), \sigma_I) = \lambda'(q', (\sigma_I, \lambda(q, \sigma_I)))$.

2.2 Automata

An automaton A is a tuple $A = (Q, q_0, \Sigma, \delta, Acc)$, where Q is a finite set of states, $q_0 \subseteq Q$ is the initial state, $\delta : Q \times \Sigma \rightarrow Q$ is the transition function, and Acc is the acceptance condition. 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 $q_{i+1} = \delta(q_i, \sigma_i)$. A accepts a trace $\overline{\sigma}$ if its run $\overline{q}$ is accepting ($Acc(\overline{q}) = \top$); its language $L(A)$ consists of the set of traces it accepts.

2.3 Acceptance conditions

The specifications we use are automata and we synthesize a system that realizes a given specification using games. Both automata and games can have the following acceptance conditions. Let Q be a set of states, an acceptance condition is a predicate $Acc : Q^\omega \rightarrow \mathbb{B}$, mapping infinite runs $\overline{q}$ to $\top$ or $\bot$ (accepting and not accepting, or winning and losing, respectively).

A *safety* acceptance condition is defined by a function $Acc(\overline{q}) = \top$ iff $\forall i \geq 0 \,.\, q_i \in F$, where $\overline{q} = q_0 q_1 \ldots$ and $F \subseteq Q$ is the set of safe states. The *reachability* acceptance condition is $Acc(\overline{q}) = \top$ iff $\exists i \geq 0 \,.\, q_i \in F$, where $F \subseteq Q$ is the set of reachable states. The *Büchi* acceptance condition is $Acc(\overline{q}) = \top$ iff $\inf(\overline{q}) \cap F \neq \emptyset$, where $F \subseteq Q$ is the set of accepting states and $\inf(\overline{q})$ is the set of states that occur infinitely often in $\overline{q}$. We abbreviate the Büchi condition as $\mathcal{B}(F)$. A *Generalized Reactivity 1* (GR(1)) acceptance condition is a predicate $\bigwedge_{i=1}^{m} \mathcal{B}(E_i) \rightarrow \bigwedge_{i=1}^{n} \mathcal{B}(F_i)$, with $E_i \subseteq Q$ and $F_i \subseteq Q$. The acceptance condition is a *generalized Büchi* acceptance condition if $m = 0$. A *Streett* acceptance condition with k pairs is a predicate $\bigwedge_{i=1}^{k} \big(\mathcal{B}(E_i) \rightarrow \mathcal{B}(F_i)\big)$.

2.4 Specifications

A *specification* φ is a set $L(\varphi) \subseteq \Sigma^\infty$ of allowed traces. A design $\mathcal{D}$ *realizes* φ, denoted by $\mathcal{D} \models \varphi$, iff $L(\mathcal{D}) \subseteq L(\varphi)$. φ is *realizable* if there exists a design $\mathcal{D}$ that realizes it.

A property φ^s defines a *safety* property [2] if finite traces that do not satisfy φ^s cannot be extended to traces that satisfy φ^s, i.e., $\forall \overline{\sigma} \in \Sigma^* .(\overline{\sigma} \not\models \varphi^s \rightarrow (\forall \overline{\sigma}' \in \Sigma^\infty .(\overline{\sigma} \cdot \overline{\sigma}') \not\models \varphi^s))$. The intuition is that a safety property states that "something bad" must never happen. If φ^s does not hold for a trace, then at some point some "bad thing" must have happened and such a "bad thing" must be irremediable. We represent a pure safety specification φ^s by a safety automaton $\varphi^s = (Q, q_0, \Sigma, \delta, F)$, where $F \subseteq Q$ is a set of safe states. A trace $\overline{\sigma}$ (of a design $\mathcal{D}$) *satisfies* φ^s if the induced run is accepting. The *language* $L(\varphi^s)$ is the set of all traces satisfying φ.

A property φ^l defines a *liveness* property [2] if every finite trace can be extended to an infinite trace that satisfies φ^l, i.e., $\forall \overline{\sigma} \in \Sigma^* \,.\, \exists \overline{\sigma}' \in \Sigma^\omega .(\overline{\sigma} \cdot \overline{\sigma}') \models \varphi^l$. Informally, a liveness property stipulates that a "good thing" happens during execution eventually.

2.5 Games

A (2-player, alternating) *game* is a tuple $\mathcal{G} = (G, g_0, \Sigma_I, \Sigma_O, \delta, Acc)$, where G is a finite set of game states, $g_0 \in G$ is the initial state, $\delta : G \times \Sigma_I \times \Sigma_O \rightarrow G$ is a complete transition function, and $Acc : G^\omega \rightarrow \mathbb{B}$ is a winning condition. 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. Thus, a finite or an infinite word over Σ results in a finite or an infinite *play*, a sequence $\overline{g} = g_0 g_1 \ldots$ of game states. A play is *won* by the system iff $Acc(\overline{g})$ is $\top$.

A deterministic (memoryless) *strategy* for the environment is a function $\rho_e : G \to \Sigma_I$. A non-det. (memoryless) *strategy* for the system is a relation $\rho_s : G \times \Sigma_I \to 2^{\Sigma_O}$ and a det. (memoryless) *strategy* for the system is a function $\rho_s : G \times \Sigma_I \to \Sigma_O$. A strategy ρ_s is *winning* for the system if, *for all* strategies ρ_e of the environment, the play $\overline{g}$ that is constructed when defining the outputs using ρ_e and ρ_s satisfies $Acc(\overline{g})$. The *winning region* W is the set of states from which a winning strategy for the system exists. A *counterstrategy* is a winning strategy for the environment from g_0. A counterstrategy exists if $g_0 \notin W$. Let $\mathcal{G} = (G, g_0, \Sigma_I, \Sigma_O, \delta, F)$ be a safety game with winning region W. If $g_0 \notin W$, a counterstrategy can be computed by solving a reachability game $\mathcal{G}' = (G, g_0, \Sigma_I, \Sigma_O, \delta, Q \backslash F)$. In safety, reachability and Büchi games, both players have memoryless winning strategies, whereas for GR(1), Streett and generalized Büchi games, finite-memory strategies are necessary for the system.

It is easy to transform a safety specification φ^s into a safety game such that a trace satisfies the specification iff the corresponding play is won. A finite trace $\overline{\sigma} \in \Sigma^*$ is *wrong* if the corresponding play $\overline{g}$ contains a state outside the winning region W. Otherwise $\overline{\sigma}$ is called *correct*. An *output* is called *wrong* if it makes a trace wrong; i.e., given φ^s, a trace $\overline{\sigma} \in \Sigma^*$, $\sigma_I \in \Sigma_I$, and $\sigma_O \in \Sigma_O$, σ_O is wrong iff $\overline{\sigma}$ is correct, but $\overline{\sigma} \cdot (\sigma_I, \sigma_O)$ is wrong. Otherwise σ_O is called *correct*.

2.6 Comparing strategies

First, we compare non-deterministic winning strategies of the system by comparing the *behaviours* that they allow [3]. If ρ is a strategy and g is a state of $\mathcal{G}$ from which ρ is winning then $Beh(\mathcal{G}, g, \rho)$ is the set of all plays starting in g and respecting ρ. If ρ is not winning from g then we put $Beh(\mathcal{G}, g, \rho) = \emptyset$. A strategy subsumes another strategy if it allows more behaviours, i.e., a strategy ρ' is *subsumed* by ρ, which is denoted $\rho' \sqsubseteq \rho$ if $Beh(\mathcal{G}, g, \rho') \subseteq Beh(\mathcal{G}, g, \rho)$ for all $g \in G$. A strategy ρ is *permissive* if $\rho' \sqsubseteq \rho$ for every memoryless strategy ρ'.

Second, we compare deterministic strategies of the system in game states from which the system cannot force a win [19]. A system strategy ρ_s is *cooperatively winning* if there exists an environment strategy ρ_e such that the play $\overline{g}$ constructed by ρ_e and ρ_s satisfies $Acc(\overline{g})$. For a Büchi game $\mathcal{G}$ with accepting states F, consider a strategy ρ_e of the environment, a strategy ρ_s of the system, and a state $g \in G$. We define the distance $dist(g, \rho_e, \rho_s) = d$ if the play $\overline{g}$ defined by ρ_e and ρ_s reaches from g an accepting state that occurs infinitely often in $\overline{g}$ in d steps. If no such state is visited, we set $dist(g, \rho_e, \rho_s) = \infty$. Given two strategies ρ_s and ρ_s' of the system, we say that ρ_s' *dominates* ρ_s if: (i) for all ρ_e and all $g \in G$, $dist(g, \rho_e, \rho_s') \leq dist(g, \rho_e, \rho_s)$, and (ii) there exists ρ_e and $g \in G$ such that $dist(g, \rho_e, \rho_s') < dist(g, \rho_e, \rho_s)$. A strategy is *admissible* if there is no strategy that dominates it.

3 Shield synthesis for reactive systems

The goal of shield synthesis is to enforce a small set of safety properties at runtime, even if these properties may be violated by the reactive system, called the *design*. We synthesize a

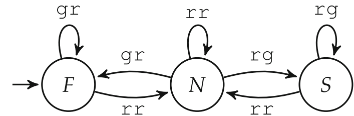


Fig. 2 Traffic light specification

Table 1 Controller shielded by $\mathcal{S}_A$

Time step	1	2	3	4	5
Controller	`rr`	**`gg`**	**`gr`**	`gr`	`rr`
Shield $\mathcal{S}_A$	`rr`	*`rg`*	*`rr`*	`gr`	`rr`

Table 2 Controller shielded by $\mathcal{S}_B$

Time step	1	2	3	4	5
Controller	`rr`	**`gg`**	`gr`	`gr`	`rr`
Shield $\mathcal{S}_B$	`rr`	*`rr`*	`gr`	`gr`	`rr`

shield directly from the set of safety properties, and attach it to the design as illustrated in Fig. 1. The shield monitors the input/output of the design and corrects the erroneous output values instantaneously, but only if necessary and as infrequently as possible. In the next section, we consider an example of a simple traffic light controller to illustrate the challenges addressed by shield synthesis.

3.1 Motivating example

Let us consider the example of a traffic light controller of two roads. There are red (`r`) or green (`g`) lights for both roads, i.e., $\Sigma = \Sigma_O = \{\texttt{rr}, \texttt{rg}, \texttt{gr}, \texttt{gg}\}$. Although the traffic light controller interface is simple, the actual implementation can be complex. The controller may have to be synchronized with other traffic lights, and it can have input sensors for cars, buttons for pedestrians, and sophisticated algorithms to optimize traffic throughput based on all sensors, the time of the day, and even the weather. As a result, the actual design may become too complex to be formally verified.

Suppose now that two safety properties are crucial and must be satisfied with certainty: (1) The output `gg`—meaning that both roads have green lights—is never allowed. (2) The output cannot change from `gr` to `rg`, or vice versa, without passing `rr`. These two properties serve as specification φ for the shield, and can be expressed by the automaton shown in Fig. 2. Edges are labeled with the controller's outputs for the two roads. F denotes the state where the first road has the green light, S denotes the state where the second road has the green light, and N denotes the state where both have red lights. There is also an error state, which is not shown. Missing edges lead to this error state, denoting forbidden situations, e.g., `gg` is never allowed. Although the automaton may not be a complete specification for a full traffic light system controller design, the corresponding shield can prevent catastrophic failures.

Tables 1 and 2 show how two different shields ($\mathcal{S}_A$ and $\mathcal{S}_B$, respectively) correct a sample output of a traffic light controller. Let us first consider Table 1. In time step 1, the controller sends the output `rr` which is accepted and passed on by the shield $\mathcal{S}_A$. In step 2, the controller sends `gg`, which violates φ. $\mathcal{S}_A$ has three options for a correction: changing the output from `gg` to either `rg`, `gr`, or `rr`. The shield $\mathcal{S}_A$ corrects the output to `rg`. In step 3, the controller gives the output `gr`. Since the traffic light cannot toggle from `rg` to `gr` according to φ, $\mathcal{S}_A$

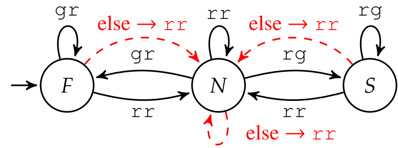


Fig. 3 Traffic light shield $\mathcal{S}_B$

changes the output to rr. Afterwards, the controller again sends gr and $\mathcal{S}_A$ is able to end the deviation and to pass on outputs from the controller until the next specification violation.

Let us analyse the behavior of the shield $\mathcal{S}_A$. First, the shield's output was correct with respect to φ. Second, to ensure minimum deviation, $\mathcal{S}_A$ only deviated from the controller when a property violation became unavoidable. Finally, the shield ended deviation after 2 steps, and then handed back control to the traffic light controller.

In Table 2 we use the shield $\mathcal{S}_B$, which corrects the controller's output in step 2 to rr. This time if the controller sends gr in step 3, the shield can give the same output as the controller immediately. If we compare the shields, $\mathcal{S}_B$ ends the deviation phase faster than $\mathcal{S}_A$. Hence, we prefer the behavior induced by $\mathcal{S}_B$.

The behavior of the shield $\mathcal{S}_B$ is illustrated in Fig. 3. Edges are labeled with the inputs of the shield. Red dashed edges denote situations where the output of the shield is different from its inputs. The modified output is written after the arrow. For all non-dashed edges, the input is just copied to the output.

The challenge in shield synthesis lies in the fact that we do not know the future inputs/outputs of the design. The question is, without knowing what the future inputs/outputs are, how should the shield correct bad behavior of the design to avoid unnecessarily large deviation in the future? For instance, the correction of shield $\mathcal{S}_A$ in step 2 was suboptimal, since it caused a deviation for 2 steps instead of 1. In the next section, we discuss a general framework of shield synthesis for reactive systems.

3.2 Definition of shields

A shield reads the input and output of a design as shown in Fig. 1. In this section, we formally define the two desired properties: *correctness* and *minimum deviation*.

3.2.1 The correctness property

By correctness, we refer to the property that the shield corrects any design's output such that a given safety specification is satisfied. Formally, let φ be a safety specification and $\mathcal{S} = (Q', q_0', \Sigma, \Sigma_O, \delta', \lambda')$ be a Mealy machine. We say that $\mathcal{S}$ *ensures correctness* if for any design $\mathcal{D} = (Q, q_0, \Sigma_I, \Sigma_O, \delta, \lambda)$, it holds that $(\mathcal{D} \circ \mathcal{S}) \models \varphi$.

Since a shield must work for any design, the synthesis procedure does not consider the design's implementation. This property is crucial because the design may be unknown or too complex to analyze. On the other hand, the design may satisfy additional (noncritical) specifications that are not specified in φ but should be retained as much as possible (i.e., as long as these additional properties are not in conflict with the critical ones).

3.2.2 The minimum deviation property

Minimum deviation requires a shield to deviate only if necessary, and as infrequently as possible. To ensure minimum deviation, a shield can only deviate from the design if a property

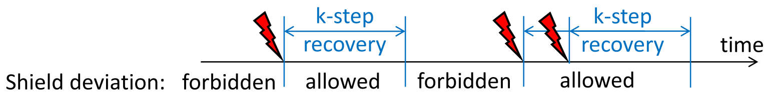


Fig. 4 Recovery phases of k-stabilizing shields

violation becomes unavoidable. Given a safety specification φ, a Mealy machine $\mathcal{S}$ *does not deviate unnecessarily* if for any design $\mathcal{D}$ and any trace $\overline{\sigma_I}||\overline{\sigma_O}$ of $\mathcal{D}$ that is not wrong, we have that $\mathcal{S}(\overline{\sigma_I}||\overline{\sigma_O}) = \overline{\sigma_O}$. In other words if $\mathcal{D}$ does not violate φ, $\mathcal{S}$ keeps the output of $\mathcal{D}$ intact.

Definition 1 Given a specification φ, a Mealy machine $\mathcal{S}$ is a *shield* if for any design $\mathcal{D}$, it holds that $(\mathcal{D} \circ \mathcal{S}) \models \varphi$ and $\mathcal{S}$ does not deviate from $\mathcal{D}$ unnecessarily.

Ideally, shields end phases of deviation as soon as possible, recovering quickly. This property leaves room for interpretation. Different types of shields differentiate on how this property is realized. In the next sections we will discuss k-stabilizing shields and admissible shields.

3.3 k-stabilizing shields

We assume that through transmission errors an arbitrary number of correct outputs by the design $\mathcal{D}$ are replaced by wrong outputs; i.e., by outputs after which a property violation becomes unavoidable (in the worst case over future inputs). After each wrong output, a k-stabilizing shield $\mathcal{S}$ enters a recovery phase and is allowed to deviate from the design's outputs for at most k consecutive time steps, including the current step. This is illustrated in Fig. 4. Wrong outputs are indicated by lightning.

We will now define k-stabilizing shields.

Definition 2 Let φ be a safety specification and let $\overline{\sigma} = (\overline{\sigma_I}||\overline{\sigma_O}) \in \Sigma^\omega$ be a correct trace. A shield $\mathcal{S}$ *adversely k-stabilizes* $\overline{\sigma}$ if, for any trace $\overline{\sigma}^f = (\overline{\sigma_I}||\overline{\sigma_O}^f) \in \Sigma^\omega$ in which for any i with $\overline{\sigma_O}[i] \neq \overline{\sigma_O}^f[i]$ it holds that $(\overline{\sigma_I}[0\ldots i-1]||\overline{\sigma_O}[0\ldots i-1]) \cdot (\overline{\sigma_I}[i], \overline{\sigma_O}[i]^f)$ is wrong and $E = \{i \mid \overline{\sigma_O}[i] \neq \overline{\sigma_O}^f[i]\}$ we have

$$\begin{aligned}
&\overline{\sigma_O}^\star = \mathcal{S}(\overline{\sigma_I}, \overline{\sigma_O}^f),\\
&(\overline{\sigma_I}||\overline{\sigma_O}^\star) \models \varphi \text{ and}\\
&\forall j \,.\, \overline{\sigma_O}^\star[j] \neq \overline{\sigma_O}^f[j] \rightarrow \exists i \in E . j - i \leq k.
\end{aligned}$$

Substituting an arbitrary number of outputs in $\overline{\sigma_O}$ by wrong outputs results in a new trace $\overline{\sigma}^f$. E denotes the indices of outputs in $\overline{\sigma}^f$ that are wrong. After any wrong output $\overline{\sigma_O}^f[i]$ with $i \in E$, the output of the shield $\overline{\sigma_O}^\star$ and the output of the design $\overline{\sigma_O}^f$ are allowed to deviate for at most k consecutive time steps.

Note that it is not always possible to adversely k-stabilize any finite trace for a given k or even for any k.

Definition 3 (*k-Stabilizing Shields*[8]) A shield $\mathcal{S}$ is k-stabilizing if it adversely k-stabilizes any finite trace.

A k-stabilizing shield guarantees to deviate from outputs of the design for at most k steps after each wrong output and to produce a correct trace. To understand the intuition behind adversely k-stabilizing a trace, suppose we take the point of view that the design

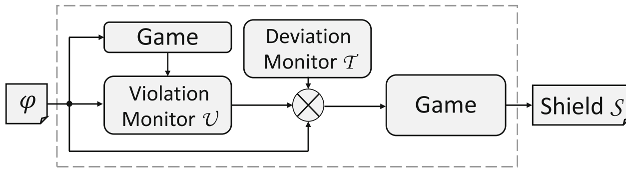
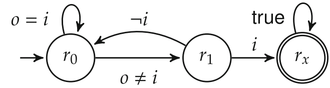


Fig. 5 Outline of the shield synthesis procedure

Fig. 6 The safety automaton φ



produces some wrong trace $\overline{\sigma}^f = (\overline{\sigma_I} || \overline{\sigma_O}^f)$ but intended to produce some correct trace $\overline{\sigma} = (\overline{\sigma_I} || \overline{\sigma_O})$. For each wrong output of $\overline{\sigma_O}^f$, the shield must "guess" what correct output the design intended in order to produce some correct trace $\overline{\sigma}^\star = (\overline{\sigma_I} || \overline{\sigma_O}^\star)$. If the shield "guesses" a particular output incorrectly, it may have to deviate from subsequent outputs of the design that would have been correct in $\overline{\sigma_O}$ in order to meet the specification. The term *adversely* k-stabilizing means that such periods of deviation will last for at most k steps for *any* intended trace $\overline{\sigma_O}$ of the design, i.e., even if $\overline{\sigma_O}^\star \neq \overline{\sigma_O}$.

3.4 Synthesizing k-stabilizing shields

The flow of our synthesis procedure for k-stabilizing shields is illustrated in Fig. 5. Let $\varphi_1, \ldots, \varphi_l$ be the critical safety properties, where each φ_i is represented as an automaton $\varphi_i = (Q_i, q_{0,i}, \Sigma, \delta_i, F_i)$. The synchronous product $\varphi = (Q, q_0, \Sigma, \delta, F)$ of these automata is again a safety automaton. Starting from φ, our shield synthesis procedure consists of five steps.

Step 1 Constructing the violation monitor $\mathcal{U}$: From φ we build an automaton $\mathcal{U} = (U, u_0, \Sigma, \delta^u)$ to monitor property violations by the design. The goal is to identify the latest point in time from which a specification violation can still be corrected with a deviation by the shield. This constitutes the start of the *recovery* phase, in which the shield is allowed to deviate from the design. The violation monitor $\mathcal{U}$ observes the design from all states the design could reach under the current input and a correct output. We note that when multiple states are being monitored, if the design's output is wrong from all monitored states, $\mathcal{U}$ monitors all states the design could reach from all currently monitored states under the current input. If the design's output is correct from one or more currently monitored states, it only continues monitoring states reachable from those monitored states under the design's output.

The first phase of the construction (Step 1-a) of $\mathcal{U}$ considers $\varphi = (Q, q_0, \Sigma, \delta, F)$ as a *safety game* and computes its winning region $W \subseteq F$ so that every reactive system $\mathcal{D} \models \varphi$ must produce outputs such that the next state of φ stays in W. Only in cases in which the next state of φ is outside of W is the shield allowed to interfere.

Example 1 Consider the safety automaton φ in Fig. 6, where i is an input, o is an output, and r_x is an unsafe state. The winning region is $W = \{r_0\}$ because from r_1 the input i controls whether r_x is visited. The shield must be allowed to deviate from the original transition $r_0 \rightarrow r_1$ if $o \neq i$. In r_1 it is too late because avoiding an unsafe state can no longer be guaranteed, given that the shield can modify the value of o but not i.

The second phase (Step 1-b) expands the state space Q to 2^Q via a subset construction with the following rationale. If $\mathcal{D}$ makes a mistake (i.e., picks outputs such that φ enters a

state $q \notin W$), the shield has to "guess" what the design actually meant to do. $\mathcal{U}$ considers all output letters that would have avoided leaving W and continues monitoring $\mathcal{D}$ from all the corresponding successor states in parallel. Thus, $\mathcal{U}$ is a subset construction of φ, where a state $u \in U$ of $\mathcal{U}$ represents a set of states in φ.

The third phase (Step 1-c) expands the state space of $\mathcal{U}$ by adding a Boolean variable d to indicate whether the shield is in the recovery period, and a Boolean output variable z. Initially d is $\bot$. Whenever there is a property violation by $\mathcal{D}$, d is set to $\top$ in the next step. If $d = \top$, the shield is in the recovery phase and can deviate. In order to decide when to set d from $\top$ to $\bot$, we add an output z to the shield. If $z = \top$ and $d = \top$, then d is set to $\bot$.

From φ, the final violation monitor is $\mathcal{U} = (U, u_0, \Sigma^u, \delta^u)$, with the states $U = (2^Q \times \{\top, \bot\})$, the initial state $u_0 = (\{q_0\}, \bot)$, the input/output alphabet $\Sigma^u = \Sigma_I \times \Sigma_O^u$ with $\Sigma_O^u = 2^{O \cup z}$, and the next-state function δ^u which obeys the following rules:

1. $\delta^u((u, d), (\sigma_I, \sigma_O)) = \big(\{q' \in W \mid \exists q \in u, \sigma_O{}' \in \Sigma_O^u \,.\, \delta(q, (\sigma_I, \sigma_O{}')) = q'\}, \top\big)$ if $\forall q \in u \,.\, \delta(q, (\sigma_I, \sigma_O)) \notin W$, and
2. $\delta^u((u, d), \sigma) = \big(\{q' \in W \mid \exists q \in u \,.\, \delta(q, \sigma) = q'\}, \mathtt{dec}(d)\big)$ if $\exists q \in u \,.\, \delta(q, \sigma) \in W$, and $\mathtt{dec}(\bot) = \bot$, and if z is $\top$ then $\mathtt{dec}(\top) = \bot$, else $\mathtt{dec}(\top) = \top$.

Our construction sets $d = \top$ whenever $\mathcal{D}$ leaves the winning region, rather than waiting until the design enters an unsafe state. Conceptually, this allows $\mathcal{S}$ to take remedial action as soon as the "the crime is committed" but before the damage actually takes place, which may be too late to correct erroneous outputs of the design.

Example 2 We illustrate the construction of $\mathcal{U}$ using the specification φ from Fig. 2, which becomes a safety automaton if we make all missing edges point to an (additional) unsafe state. The winning region consists of all safe states, i.e., $W = \{F, N, S\}$. The resulting violation monitor $\mathcal{U}$ is illustrated in Fig. 8. In this example, z is always set to $\top$. For the sake of clarity, z is not shown. The update of d is as follows: Whenever the design commits a violation (indicated by red dashed edges), then d is set to $\top$. Otherwise, d is set to $\bot$.

Let us take a closer look at some of the edges of $\mathcal{U}$ in Fig. 8. If the current state is $(\{F\}, \bot)$ and $\mathcal{U}$ observes the output gg from the design, a specification violation occurs. We assume that $\mathcal{D}$ meant to give an allowed output, i.e., either gr or rr. $\mathcal{U}$ continues to monitor both F and N; thus, $\mathcal{U}$ enters the state $(\{F, N\}, \top)$. If the next observation is again gg, which is neither allowed in F nor in N, we know that a second violation occurred. $\mathcal{U}$ continues to monitor the design from all states that are reachable from the current set of monitored states: in this case all three states and $\mathcal{U}$ enters the state $(\{F, N, S\}, \top)$. If the next observation is rr, then d is set to $\bot$ and $\mathcal{U}$ enters the state $(\{F\}, \bot)$. This constitutes the end of the recovery period

Step 2 Constructing the deviation monitor $\mathcal{T}$: We build $\mathcal{T} = (T, t_0, \Sigma_O \times \Sigma_O, \delta^t)$ to monitor deviations between the shield's and design's outputs. Here, $T = \{t_0, t_1\}$ and $\delta^t(t, (\sigma_O, \sigma_O{}')) = t_0$ iff $\sigma_O = \sigma_O{}'$. That is, if there is a deviation in the current time step, then $\mathcal{T}$ will be in t_1 in the next time step. Otherwise, it will be in t_0. This deviation monitor is shown in Fig. 7.

Step 3 Constructing and solving the safety game $\mathcal{G}^s$. We construct a safety game $\mathcal{G}^s$ such that any shield that implements a winning strategy for $\mathcal{G}^s$ is allowed to deviate in the recovery period only, and the output of the shield is always correct.

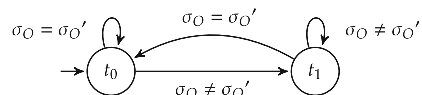


Fig. 7 The deviation monitor $\mathcal{T}$

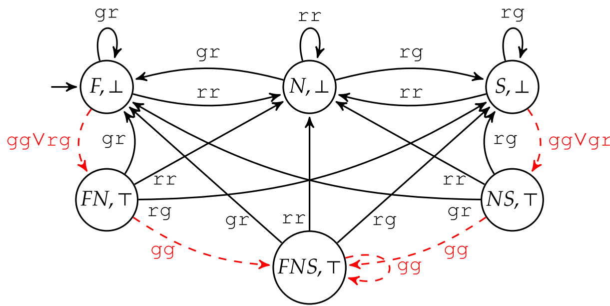


Fig. 8 Violation monitor $\mathcal{U}$ of Example 2

Let the automata $\mathcal{U}$ and $\mathcal{T}$ and the safety automaton φ be given. Let $W \subseteq F$ be the winning region of φ when considered as a safety game. We construct a safety game $\mathcal{G}^s = (G^s, g_0^s, \Sigma_I^s, \Sigma_O^s\ \delta^s, F^s)$, which is the synchronous product of $\mathcal{U}$, $\mathcal{T}$ and φ such that $G^s = U \times T \times Q$ is the state space, $g_0^s = (u_0, t_0, q_0)$ is the initial state, $\Sigma_I^s = \Sigma_I \times \Sigma_O$ is the input alphabet, $\Sigma_O^s = \Sigma_O$ is the output alphabet, δ^s is the next-state function, and F^s is the set of safe states, such that $\delta^s\big((u, t, q), (\sigma_I, \sigma_O), \sigma_O{}'\big) =$

$$\big(\delta^u(u, (\sigma_I, \sigma_O)), \delta^t(t, (\sigma_O, \sigma_O{}')), \delta(q, (\sigma_I, \sigma_O{}'))\big),$$

and $F^s = \{(u, t, q) \in G^s \mid ((q \in W) \wedge (u = (w \in 2^W, \bot) \rightarrow t = t_0))\}$.

In the definition of F^s, we require that $q \in W$, i.e., it is a state of the winning region, which ensures that the shield output will satisfy φ. The second term ensures that the shield can only deviate in the recovery period, i.e., while $d = \top$ in $\mathcal{U}$.

We use standard algorithms for safety games (cf. [19]) to compute the winning region W^s and the permissive winning strategy $\rho_s : G \times \Sigma_I \rightarrow 2^{\Sigma_O}$ that is not only winning for the system, but also subsumes all memoryless winning strategies.

Step 4 Constructing the Büchi game $\mathcal{G}^b$. A shield $\mathcal{S}$ that implements the winning strategy ρ_s of the safety game ensures correctness ($\mathcal{D} \circ \mathcal{S} \models \varphi$) and keeps the output of the design $\mathcal{D}$ intact if $\mathcal{D}$ does not violate φ. What is still missing is to keep the number of deviations per violation to a minimum. As a basic requirement, we would like the recovery period to be over infinitely often. We will see later (in Theorem 1) that this basic requirement is enough to ensure not only a finite recovery period but also the shortest possible recovery period. This requirement can be formalized as a Büchi winning condition. We construct the Büchi game $\mathcal{G}^b$ by applying the permissive safety strategy ρ^s to the game graph $\mathcal{G}^s$.

Given the safety game $\mathcal{G}^s = (G^s, g_0^s, \Sigma_I^s, \Sigma_O^s, \delta^s, F^s)$ with the non-deterministic winning strategy ρ^s and the winning region W^s, we construct a Büchi game $\mathcal{G}^b = (G^b, g_0^b, \Sigma_I^b, \Sigma_O^b, \delta^b, F^b)$ such that $G^b = W^s$ is the state space, the initial state $g_0^b = g_0^s$ and the input/output alphabet $\Sigma_I^b = \Sigma_I^s$ and $\Sigma_O^b = \Sigma_O^s$ remain unchanged, $\delta^b = \delta^s \cap \rho^s$ is the transition function, and $F^b = \{(u, t, q) \in W^s \mid u = (w \in 2^W, \bot)\}$ is the set of accepting states. A play is winning if $d = \bot$ infinitely often.

Step 5 Solving the Büchi game $\mathcal{G}^b$. We use standard algorithms for Büchi games (cf. e.g. [30]) to compute a winning strategy ρ^b for $\mathcal{G}^b$. If a winning strategy exists, we implement this strategy in a new reactive system $\mathcal{S} = (G^b, g_0^b, \Sigma_I^b, \Sigma_O^b, \delta', \rho^b)$ with $\delta'(g, \sigma_I) = \delta^b(g, \sigma_I, \rho^b(g, \sigma_I))$.

Theorem 1 *A system that implements the winning strategy* ρ^b *in the Büchi game* $\mathcal{G}^b = (G^b, g_0^b, \Sigma_I^b, \Sigma_O^b, \delta^b, F^b)$ *in a new reactive system* $\mathcal{S} = (G^b, g_0^b, \Sigma_I^b, \Sigma_O^b, \delta', \rho^b)$ *with* $\delta'(g, \sigma_I) = \delta^b(g, \sigma_I, \rho^b(g, \sigma_I))$ *is a k-stabilizing shield for the smallest k possible.*

Proof Since $\rho^b \sqsubseteq \rho^s$, implementing ρ^b ensures correctness ($\mathcal{D} \circ \mathcal{S} \models \varphi$) and that $\mathcal{S}$ does not deviate from $\mathcal{D}$ unnecessarily. Therefore, $\mathcal{S}$ is a shield (see Definition 1). Furthermore, the strategy ρ^b ensures that the recovery period is over infinitely often. Since winning strategies for Büchi games are subgame optimal, a shield that implements ρ^b ends deviations at any state after the smallest number of steps possible, i.e., $\mathcal{S}$ adversely k-stabilizes any trace for the smallest k possible. Hence, $\mathcal{S}$ is a k-stabilizing shield (see Definition 3). □

The standard algorithm for solving Büchi games contains the computation of attractors. The i-th attractor for the system contains all states from which the system can "force" a visit of an accepting state in i steps. For all states $g \in G^b$ of the game $\mathcal{G}^b$, the attractor number i of g corresponds to the smallest number of steps within which the recovery phase can be guaranteed to end.

Theorem 2 *Let* $\varphi = \{Q, q_0, \Sigma, \delta, F\}$ *be a safety specification and* $|Q|$ *be the cardinality of the state space of* φ. *A k-stabilizing shield with respect to* φ *can be synthesized in* $\mathcal{O}(|Q|^3 2^{|Q|})$ *time if it exists.*

Proof Our safety game $\mathcal{G}^s$ and our Büchi game $\mathcal{G}^b$ have at most $m = (2 \cdot 2^{|Q|}) \cdot 2 \cdot |Q|$ states and at most $n = m^2$ edges. Safety games can be solved in $\mathcal{O}(m+n)$ time and Büchi games in $\mathcal{O}(m \cdot n)$ time [30]. □

While an exponential runtime may not look appealing at the first glance, keep in mind that the critical safety properties of a system are typically simple and that the complexity of the design is irrelevant for the shield synthesis procedure.

3.5 Admissible shields

In this section we define admissible shields. We distinguish between two situations. In states of the design in which a finite number k of deviations can be guaranteed, an admissible shield takes an adversarial view on the design: it guarantees recovery within k steps regardless of system behavior, for the smallest k possible. In these states, the strategy of an admissible shield conforms to the strategy of a k-stabilizing shield. In all other states, admissible shields take a collaborative view: the admissible shield will attempt to work with the design to recover as soon as possible. In particular, an admissible shield plays an admissible strategy, that is, a strategy that cannot be beaten in recovery speed if the design acts cooperatively. We will now define admissible shields.

Definition 4 *(Adversely subgame optimal shield)* A shield $\mathcal{S}$ is *adversely subgame optimal* if, for any trace $\overline{\sigma} \in \Sigma^*$, $\mathcal{S}$ adversely $k-$stabilizes $\overline{\sigma}$ (Definition 2) and there exists no shield that adversely l-stabilizes $\overline{\sigma}$ for any $l < k$.

An adversely subgame optimal shield $\mathcal{S}$ guarantees to deviate in response to an error for at most k time steps, for the smallest k possible.

Definition 5 Let φ be a safety specification and let $\overline{\sigma} = (\overline{\sigma_I} || \overline{\sigma_O}) \in \Sigma^\omega$ be a correct trace. Let $\overline{\sigma}^f = (\overline{\sigma_I} || \overline{\sigma_O}^f) \in \Sigma^\omega$ be a trace in which $\forall i$ with $\overline{\sigma_O}[i] \neq \overline{\sigma_O}^f[i]$ it holds that $\overline{\sigma}[0 \ldots i-1] \cdot (\overline{\sigma_I}[i], \overline{\sigma_O}[i]^f)$ is wrong and let $E = \{i \mid \overline{\sigma_O}[i] \neq \overline{\sigma_O}^f[i]\}$. A shield $\mathcal{S}$

collaboratively k-stabilizes $\overline{\sigma}$ if for any wrong output $\overline{\sigma_O}^f[i]$ with $i \in E$ the following holds: For any correct output $\sigma_O{}^c \in \Sigma_O$ (i.e., $\overline{\sigma}[0\ldots i-1]\cdot(\overline{\sigma_I}[i], \sigma_O{}^c)$ is correct), *there exists* a correct trace $(\overline{\sigma_I}^c||\overline{\sigma_O}^c) \in \Sigma^\omega$ (i.e., $\overline{\sigma}[0\ldots i-1]\cdot(\overline{\sigma_I}[i], \sigma_O{}^c)\cdot(\overline{\sigma_I}^c||\overline{\sigma_O}^c)$ is correct), such that

$$\overline{\sigma}^{\#} := \overline{\sigma}[0\ldots i]^f \cdot (\overline{\sigma_I}^c||\overline{\sigma_O}^c), \overline{\sigma_O}^\star := \mathcal{S}(\overline{\sigma}^{\#}),$$
$$(\overline{\sigma_I}||\overline{\sigma_O}^\star) \models \varphi \text{ and } \forall j \geq i\,.\,\overline{\sigma_O}^\star[j] \neq \overline{\sigma_O}^{\#}[j] \rightarrow j-i \leq k.$$

The trace $\overline{\sigma}^f$ results from substituting outputs in $\overline{\sigma}$ by wrong outputs, and E contains the indices of the wrong outputs as defined in Sect. 3.3. The shield has to correct any wrong output $\overline{\sigma_O}^f[i]$ with $i \in E$ with an output such that, for *some* correct output $\sigma_O{}^c$ and *some* correct continuation $(\overline{\sigma_I}^c||\overline{\sigma_O}^c)$, the shield is able to end the deviation after k-steps, and the shielded trace satisfies φ.

Definition 6 (*Collaborative k-Stabilizing Shield*) A shield $\mathcal{S}$ is collaboratively k-stabilizing if it collaboratively k-stabilizes any finite trace.

A collaborative k-stabilizing shield requires that it must be possible to end deviations after k steps, for some future input and output of $\mathcal{D}$. It is not necessary that this is possible for all future behavior of $\mathcal{D}$ allowing infinitely long deviations.

Definition 7 *(Collaborative subgame optimal shield)* A shield $\mathcal{S}$ is *collaborative subgame optimal* if, for any trace $\overline{\sigma} \in \Sigma^*$, $\mathcal{S}$ collaboratively $k-$stabilizes $\overline{\sigma}$ and there exists no shield that adversely l-stabilizes $\overline{\sigma}$ for any $l < k$.

Definition 8 *(Admissible Shield)* A shield $\mathcal{S}$ is admissible if, for any trace $\overline{\sigma}$, whenever there exists a k and a shield $\mathcal{S}'$ such that $\mathcal{S}'$ adversely k-stabilizes $\overline{\sigma}$, then $\mathcal{S}$ is an adversely subgame optimal shield and adversely k-stabilizes $\overline{\sigma}$ for a minimal k. If such a k does not exist for trace $\overline{\sigma}$, then $\mathcal{S}$ acts as a collaborative subgame optimal shield and collaboratively k-stabilizes $\overline{\sigma}$ for a minimal k.

An admissible shield ends deviations as soon as possible. In all states of the design $\mathcal{D}$ from which it is possible to k-adversely stabilize traces, an admissible shield does this for the smallest k possible. In all other states, the shield corrects the output in such a way that there exists design's inputs and outputs such that deviations end after l steps, for the smallest l possible.

3.6 Synthesizing admissible shields

The flow of the synthesis procedure for admissible shields is similar to the flow for synthesizing k-stabilizing shields, and is illustrated in Fig. 5. In order to synthesize an admissible shield, a Büchi game $\mathcal{G}^b$ is constructed in the same way as for k-stabilizing shields. The difference lies in the computation of the strategy of the Büchi game $\mathcal{G}^b$: for k-stabilizing shields we compute a winning strategy of $\mathcal{G}^b$, and for admissible shields we compute an admissible strategy. Given is a safety specification $\varphi = \{\varphi_1, \ldots, \varphi_l\} = (Q, q_0, \Sigma_I \times \Sigma_O, \delta, F)$. Starting from φ, our shield synthesis procedure is as follows:
Steps 1–4 Perform as in Sect. 3.4.
Step 5 Solving the Büchi game $\mathcal{G}^b$. A Büchi game $\mathcal{G}^b$ may contain reachable states for which $d = \bot$ cannot be enforced infinitely often, i.e., states from which a recovery in a finite time cannot be guaranteed. We implement an admissible strategy that visits states with $d = \bot$

infinitely often whenever possible. This criterion essentially asks for a strategy that is winning with the help of the design.

The admissible strategy ρ^b for a Büchi game $\mathcal{G}^b = (G^b, g_0^b, \Sigma_I^b, \Sigma_O^b, \delta^b, F^b)$ can be computed as follows[19]:

1. Compute the winning region W^b and a winning strategy ρ_w^b for $\mathcal{G}^b$ (cf. [30]).
2. Remove all transitions that start in W^b and do not belong to ρ_w^b from $\mathcal{G}^b$. This results in a new Büchi game $\mathcal{G}_1^b = (G^b, g_0^b, \Sigma_I^b, \Sigma_O^b, \delta_1^b, F^b)$ with $(g, (\sigma_I, \sigma_O), g') \in \delta_1^b$ if $(g, \sigma_I, \sigma_O) \in \rho_w^b$ or if $\forall \sigma_O' \in \Sigma_O^b .(g, \sigma_I, \sigma_O') \notin \rho_w^b \wedge (g, (\sigma_I, \sigma_O), g') \in \delta^b$.
3. In the resulting game $\mathcal{G}_1^b$, compute a cooperatively winning strategy ρ^b. In order to compute ρ^b, one first has to transform all input variables to output variables. This results in the Büchi game $\mathcal{G}_2^b = (G^b, g_0^b, \emptyset, \Sigma_I^b \times \Sigma_O^b, \delta_1^b, F^b)$. Afterwards, ρ^b can be computed with the standard algorithm for the winning strategy on $\mathcal{G}_2^b$.

The strategy ρ^b is an admissible strategy of the game $\mathcal{G}^b$, since it is winning and cooperatively winning [19]. Whenever the game $\mathcal{G}^b$ starts in a state of the winning region W^b, any play created by ρ_w^b is winning. Since ρ^b coincides with ρ_w^b in all states of the winning region W^b, ρ^b is winning. We know that ρ^b is cooperatively winning in the game $\mathcal{G}_1^b$. A proof that ρ^b is also cooperatively winning in the original game $\mathcal{G}^b$ can be found in [19].

Theorem 3 *A shield that implements the admissible strategy ρ^b in the Büchi game $\mathcal{G}^b = (G^b, g_0^b, \Sigma_I^b, \Sigma_O^b, \delta^b, F^b)$ in a new reactive system $\mathcal{S} = (G^b, g_0^b, \Sigma_I^b, \Sigma_O^b, \delta', \rho^b)$ with $\delta'(g, \sigma_I) = \delta^b(g, \sigma_I, \rho^b(g, \sigma_I))$ is an admissible shield.*

Proof Since $\rho^b \sqsubseteq \rho^s$, $\mathcal{S}$ is a shield according to Definition 1. $\mathcal{S}$ is an adversely subgame optimal shield (see Definition 4) for all states of the design in which a finite number of deviations can be guaranteed, since ρ^b is winning for all winning states of the Büchi game $\mathcal{G}^b$, and winning strategies for Büchi games are subgame optimal. Furthermore, $\mathcal{S}$ is a collaborative subgame optimal shield (see Definition 7), since ρ^b is cooperatively winning in the Büchi game $\mathcal{G}^b$ and cooperative winning strategies for Büchi games are subgame optimal for some inputs. Therefore, $\mathcal{S}$ is an admissible shield (see Definition 8). □

3.7 Shields with a fail-safe mode

When a property violation becomes unavoidable, k-stabilizing shields and admissible shields enter a recovery phase, where the shield is allowed to deviate. In the case that it can be assumed that specification violations are rare events, the construction of k-stabilizing shields and admissible shields can be modified to tolerate the next violation only after the recovery phase. If a second violation happens within the recovery period, the shield enters a fail-safe mode, where it enforces the specification, but stops minimizing the deviations from that point on, i.e., it loses the ability to recover.

For many practical examples, this modification speeds up the synthesis process and results in significantly smaller implementations of $\mathcal{S}$. This is because the set of states to be monitored by $\mathcal{U}$ generally becomes smaller. In the case of a violation, a shield enters its recovery phase and $\mathcal{U}$ observes $\mathcal{D}$ from all states that $\mathcal{D}$ could reach under the current input and a correct output. Consider the case of a second violation within the recovery phase. For shields without a fail-safe mode, $\mathcal{U}$ has to monitor the set of all input-enabled states that are reachable from the current set of monitored states. For shields with a fail-safe mode, $\mathcal{U}$ just enters a single special fail-safe state.

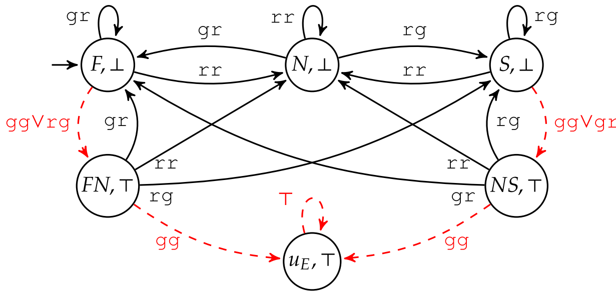


Fig. 9 Violation monitor $\mathcal{U}$ of Example 2 with a fail-safe mode

3.7.1 Construction of shields with a fail-safe mode

In order to construct a k-stabilizing shield or an admissible shield with a fail-safe mode, only small changes in the construction of the violation monitor and the Büchi game are necessary.

Changes in Step 1 Constructing the violation monitor $\mathcal{U}$: Only the third phase of the construction of $\mathcal{U}$ (see Sect. 3.4, Step 1-c) needs to be modified.

As before, we expand the state space of $\mathcal{U}$ by adding a variable d to indicate whether the shield is in the recovery phase. Additionally, we now add a special fail-safe state u_E to indicate whether the shield is in the fail-safe mode. If a second violation happens while $d = \top$, then the shield enters the fail-safe mode.

From φ, the final violation monitor is $\mathcal{U} = (U, u_0, \Sigma^u, \delta^u)$, with the set of states $U = (2^Q \cup u_E) \times \{\top, \bot\}$, $u_0 = (\{q_0\}, \bot)$, the input/output alphabet $\Sigma^u = \Sigma_I \times \Sigma_O^u$ with $\Sigma_O^u = 2^{O \cup z}$, and the next-state function δ^u that obeys the following rules:

1. $\delta^u((u_E, \top), \sigma) = (u_E, \top)$ (meaning that u_E is a trap state),
2. $\delta^u((u, \top), \sigma) = (u_E, \top)$ if $\forall q \in u \,.\, \delta(q, \sigma) \notin W$,
3. $\delta^u((u, \bot), (\sigma_I, \sigma_O)) = \big(\{q' \in W \mid \exists q \in u, \sigma_O{}' \in \Sigma_O^u \,.\, \delta(q, (\sigma_I, \sigma_O{}')) = q'\}, \top\big)$ if $\forall q \in u \,.\, \delta(q, (\sigma_I, \sigma_O)) \notin W$, and
4. $\delta^u((u, d), \sigma) = \big(\{q' \in W \mid \exists q \in u \,.\, \delta(q, \sigma) = q'\}, \mathtt{dec}(d)\big)$ if $\exists q \in u \,.\, \delta(q, \sigma) \in W$, and $\mathtt{dec}(\bot) = \bot$, and if z is $\top$ then $\mathtt{dec}(\top) = \bot$, else $\mathtt{dec}(\top) = \top$.

Example 3 Figure 9 shows $\mathcal{U}$ with a fail-safe mode using the specification φ from Fig. 2. If $d = \top$ at the violation, the next state is u_E. Otherwise, d is set to $\bot$.

Changes in Step 4 Constructing the Büchi game $\mathcal{G}^b$: We change the original construction of the Büchi game of Sect. 3.4 only slightly by adding the state $(u_E, \top)$ to the set of accepting states. The intuition is that, if the shield enters the state $(u_E, \top)$, it can stop minimizing the deviations, and the Büchi condition is trivially satisfied.

3.8 Liveness-preserving shields

Reactive systems usually not only satisfy safety properties, but are also expected to satisfy liveness properties [2], which guarantee that certain good events eventually happen. Unfortunately, it is not guaranteed that the corrections of the shield preserve the liveness properties satisfied by the design (without shielding).

Definition 9 *(Liveness-Preserving Shield)* A shield preserves a given set of liveness properties if any liveness properties satisfied by the design without shielding are also satisfied by the shield.

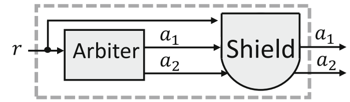


Fig. 10 Attaching the shield (Example 4)

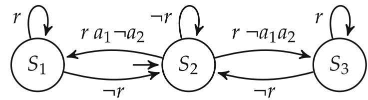


Fig. 11 Safety specification φ^s (Example 4)

Table 3 Shield $\mathcal{S}$ correcting the arbiter (Example 4)

Time step	1	2	3	4	5
Arbiter	$\neg r \neg a_1 \neg a_2$	$\boldsymbol{r\, a_1 a_2}$	$\neg r \neg a_1 \neg a_2$	$\boldsymbol{r\, a_1 a_2}$	...
Shield	$\neg r \neg a_1 \neg a_2$	$r\, \neg a_1 a_2$	$\neg r \neg a_1 \neg a_2$	$r\, \neg a_1 a_2$	...

We now give a example where $\mathcal{S}$ destroys a liveness property satisfied by $\mathcal{D}$.

Example 4 Consider a simple arbiter, with one input signal r, with which clients request permissions, and two output signals a_1 and a_2 to grant resource 1 and resource 2. The implementation of the arbiter is already given, but the full specification of the arbiter is unknown. Suppose we know that the arbiter satisfies the following two liveness properties:

– φ_1^l: Resource 1 has to be granted infinitely often, i.e., $\mathsf{G}\,\mathsf{F}(a_1)$ in LTL [32].
– φ_2^l: Resource 2 has to be granted infinitely often, i.e., $\mathsf{G}\,\mathsf{F}(a_2)$ in LTL.

We attach a shield to the arbiter as shown in Fig. 10 to enforce the safety property φ^s expressed by the safety automaton in Fig. 11. Although this safety automaton is not a complete specification of the arbiter, the corresponding shield can enforce important properties of the arbiter. φ^s states that if there is a request r in state S_2, then one resource has to be granted immediately: either resource 1 with $a_1 \neg a_2$, or resource 2 with $\neg a_1 a_2$. There is also an error state, which is not shown. Missing edges lead to this error state, e.g., granting both resources (with $a_1 a_2$) or no resource (with $\neg a_1 \neg a_2$) after a request r in state S_2 is never allowed.

Table 3 shows how $\mathcal{S}$ may correct the arbiter. In step 1, we are S_2 and the arbiter receives no request (i.e., $\neg r$). In this case, every possible output from the arbiter is accepted by the shield. In step 2, the arbiter receives a request (i.e., r), but grants both resources at once (i.e., $a_1 a_2$), which violates φ^s. The shield corrects the output to $\neg a_1 a_2$. In step 3, the arbiter receives $\neg r$, and sets all outputs to $\perp$. The shield accepts the output, and ends the deviation. From there on, everything repeats infinitely often.

Let us analyse the corrections of the shield with respect to the liveness properties φ_1^l and φ_2^l. The arbiter gave a_1 and a_2 infinitely often, thereby satisfying both φ_1^l and φ_2^l. The output of the shield however never included the symbol a_1. Although the arbiter satisfied all liveness properties, through the correction of the shield, the first liveness property φ_1^l is violated.

Next, we discuss an extension to the k-stabilizing and the admissible shield synthesis procedure that allows liveness-preserving shielding.

3.8.1 Synthesizing liveness-preserving shields

To construct a system that has all the properties of a shield (k-stabilizing or admissible) and is liveness-preserving, we create and solve a Streett game with two pairs. The first Streett

pair, called the *shielding pair*, encodes that the recovery phase has to end infinitely often. The second Streett pair, called the *liveness-preservation pair*, encodes that if the design satisfies all liveness properties, then the shield has to preserve all liveness properties.

Let $\varphi^s = \{\varphi^s_1, \ldots, \varphi^s_m\} = (Q^s, q^s_0, \Sigma^s, \delta^s, F^s)$ be the safety specification to be enforced by the shield. Let $\varphi^l = \{\varphi^l_1, \ldots, \varphi^l_n\}$ be the set of liveness properties that if satisfied by the design, have to be preserved after shielding. Each φ^l_i is represented as a Büchi automaton $\varphi^l_i = (Q^l_i, q^l_{0,i}, \Sigma^l, \delta^l_i, F^l_i)$, with F^l_i the set of states that have to be visited infinitely often. The synchronous product φ^l of $\{\varphi^l_1, \ldots, \varphi^l_n\}$ defines an automaton with generalized Büchi acceptance condition $\varphi^l = (Q^l, q^l_0, \Sigma^l, \delta^l, \{F^l_1, \ldots, F^l_n\})$. Given φ^s and φ^l, our shield synthesis procedure consists of three steps.

Step 1 Encode the shielding Streett pair $\langle \mathbf{E_1}, \mathbf{F_1} \rangle$. Given the safety specification φ^s, we construct a one-pair Streett game $\mathcal{G}^{s1}$. In the case of k-stabilizing shields, we construct $\mathcal{G}^{s1}$ in such a way that the winning strategy corresponds to a k-stabilizing shield. In case of admissible shields, we construct $\mathcal{G}^{s1}$ such that the admissible strategy implements an admissible shield.

First (Step 1-a), we construct a Büchi game $\mathcal{G}^b = (G^b, g^b_0, \Sigma_I \times \Sigma_O, \Sigma_O, \delta^b, F^b)$, using the construction described in Sect. 3.4 for k-stabilizing shields, or the construction of Sect. 3.6 for admissible shields. In both cases, the transition relation δ^b contains only transitions in which the output of the shield satisfies φ^s and there are no illegal deviations between the output of the shield and the output of the design. F^b covers all states with $d = \bot$; i.e. if these states are visited infinitely often, the recovery phase will be over infinitely often.

Next (Step 1-b), we transform the Büchi game $\mathcal{G}^b$ into the Streett game $\mathcal{G}^{s1} = (G^{s1}, g^{s1}_0, \Sigma_I \times \Sigma_O, \Sigma_O, \delta^{s1}, \langle E_1, F_1 \rangle)$ such that $G^{s1} = G^b$, $g^{s1}_0 = g^b_0$, $\delta^{s1} = \delta^b$ and $\langle E_1, F_1 \rangle = \langle G^{s1}, F^b \rangle$. The intuition is that the states in $F_1 = F^b$ must always be visited infinitely often (the deviation phase should end infinitely often); therefore, we set E_1 to the set of all states.

Step 2 Encode the liveness-preservation Streett pair $\langle \mathbf{E_2}, \mathbf{F_2} \rangle$. From the liveness specification $\varphi^l = (Q^l, q^l_0, \Sigma^l, \delta^l, \{F^l_1, \ldots, F^l_n\})$, we construct another one-pair Streett game $\mathcal{G}^{s2}$ such that a winning strategy in this game corresponds to a liveness-preserving implementation. Therefore, we turn the condition that if the design satisfies φ^l, then the shield has to satisfy φ^l as well, into the Liveness-Preservation Streett pair $\langle \mathbf{E_2}, \mathbf{F_2} \rangle$. The construction of $\mathcal{G}^{s2}$ consists of two steps.

In the first phase (Step 2-a), we create a GR(1) game $\mathcal{G}^g = (G^g, g^g_0, \Sigma_I \times \Sigma_O, \Sigma_O, \delta^g, Acc)$, such that $G^g = Q^l \times Q^l$ is the state space, $g_0 = (q^l_0, q^l_0)$ is the initial state, $\Sigma_I \times \Sigma_O$ is the input, Σ_O is the output, δ^g is the next-state function, and Acc is the GR(1) acceptance condition such that $\delta^g\big((q, q'), (\sigma_I, \sigma_O), \sigma_O{}'\big) =$

$$\big(\delta^l(q, (\sigma_I, \sigma_O)), \delta^l(q', (\sigma_I, \sigma_O{}'))\big),$$

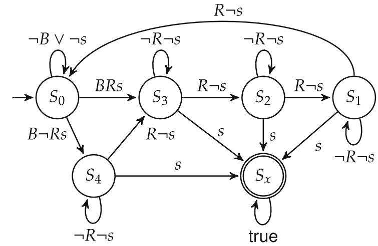
and $Acc = \bigwedge_i \inf(\overline{q}) \wedge F^l_i \neq \emptyset \rightarrow \bigwedge_i \inf(\overline{q'}) \wedge F^l_i \neq \emptyset$ with $\overline{g^g} = \overline{q} || \overline{q'} = (q_0 q'_0)(q_1 q'_1) \cdots$.

In the second phase (Step 2-b), the GR(1) game $\mathcal{G}^g$ is transformed into a one-pair Streett game $\mathcal{G}^{s2} = (G^{s2}, g^{s2}_0, \Sigma_I \times \Sigma_O, \Sigma_O, \delta^{s2}, \langle E_2, F_2 \rangle)$ via a counting construction [5].

Step 3: Construct and solve the two-pair Streett game From $\mathcal{G}^{s1}$ and $\mathcal{G}^{s2}$, we construct a Streett game $\mathcal{G}^{st}$ with two Streett pairs: $\mathcal{G}^{st} = (G^{st}, g^{st}_0, \Sigma_I \times \Sigma_O, \Sigma_O, \delta^{st}, Acc)$ with the state space $G^{st} = G^{s1} \times G^{s2}$, the initial state $g^{st}_0 = (g^{s1}_0, g^{s2}_0)$, the next-state function δ^{st} and the acceptance condition Acc, such that $\delta^{st}\big((g^{s1}, g^{s2}), (\sigma_I, \sigma_O), \sigma_O{}'\big) =$

$$\big(\delta^{s1}(g^{s1}, (\sigma_I, \sigma_O)), \delta^{s2}(g^{s2}, (\sigma_I, \sigma_O{}'))\big),$$

Fig. 12 Guarantee 3 from [7]



and $Acc = \{\langle E_1, F_1\rangle, \langle E_2, F_2\rangle\}$. A winning strategy for the two-pair Streett game $\mathcal{G}^{st}$ corresponds to a liveness-preserving k-stabilizing shield. The admissible strategy for $\mathcal{G}^{st}$ implements an admissible shield. Streett games with n Streett pairs can be solved using the recursive fixpoint algorithm of [31].

3.9 Experimental results

We implemented our k-stabilizing and admissible shield synthesis procedures in a Python tool that takes a set of safety automata defined in a textual representation as input. The first step in our synthesis procedure is to build the product of all safety automata and construct the violation monitor (see Sects. 3.4 and 3.6). This step is performed on an explicit representation. For the remaining steps we use Binary Decision Diagrams (BDDs) [10] for symbolic representation.

We have conducted two sets of experiments, where the benchmarks are (1) selected properties for an ARM AMBA bus arbiter [7], and (2) selected properties from LTL specification patterns [16]. The source code of our proof-of-concept synthesis tool as well as the input files and instructions to reproduce our experiments are available for download[1]. All experiments were performed on a machine with an Intel i5-3320M CPU@2.6 GHz, 8 GB RAM, and a 64-bit Linux.

3.9.1 ARM AMBA bus arbiter example

We used properties of an ARM AMBA bus arbiter [7] as input to our shield synthesis tool. We present the result on one example property, and then present the performance results for other properties. The property that we enforced was Guarantee 3 from the specification of [7], which says that if a length-four locked burst access starts, no other access can start until the end of this burst. The safety automaton is shown in Fig. 12, where B, R, and s are short for `hmastlock` $\wedge$ `HBURST=BURST4`, `HREADY`, and `start`, respectively. Upper case signal names are inputs, and lower-case names are outputs of the arbiter. The state S_x is unsafe. S_0 is the idle state waiting for a burst to start ($B \wedge s$). The burst is over if input R has been $\top$ 4 times. State S_i, where $i = 1, 2, 3, 4$, means that R must be $\top$ for i more times. The counting includes the time step where the burst starts, i.e., where S_0 is left. Outside of S_0, s is required to be $\bot$.

Our tool generates a 1-stabilizing shield and an admissible shield within a fraction of a second. The 1-stabilizing shield has 8 latches and 142 (2-input) multiplexers, which is then reduced by ABC [9] to 4 latches and 77 AIG gates. The admissible shield has 9 latches and 340 multiplexers, which is reduced by ABC to 7 latches and 271 AIG gates. We verified it against an arbiter implementation for 2 bus masters, where we introduced the following

[1] http://www.iaik.tugraz.at/content/research/design_verification/others/ .

Table 4 Shield execution results

Step	3	4	5	6	7	8	9	10	11	12
State in Fig. 12	S_0	S_4	S_3	S_2	S_1	S_0	S_0	S_0	S_0	…
State in Arbiter	S_0	S_3	S_2	S_1	S_0	S_3	S_2	S_1	S_0	…
B	1	1	1	1	1	1	1	1	1	…
R	0	1	1	1	1	1	1	1	1	…
s from Arbiter	1	0	0	0	1↯	0	0	0	0	…
s from Shield	1	0	0	0	0	0	0	0	0	…

Table 5 Performance for AMBA [7]

Property	$\|Q\|$	$\|I\|$	$\|O\|$	$\mathcal{S}$ with fail-safe mode		$\mathcal{S}$ w/o fail-safe mode	
				k	Time (s)	k	Time (s)
P1:G1	3	1	1	1	0.1	2	0.1
P2:G1 + 2	5	3	3	1	0.1	2	0.6
P3:G1 + 2 + 3	12	3	3	1	0.1	5	0.26
P4:G1 + 2 + 4	8	3	6	2	7.8	2	12
P5:G1 + 3 + 4	15	3	5	2	65	5	117
P6:G1 + 2 + 3 + 5	18	3	4	2	242	5	325
P7:G1 + 2 + 4 + 5	12	3	7	1	45 (admissible)	2	66 (admissible)
P8:G2 + 3 + 4	17	3	6	2	98 (admissible)	5	129 (admissible)
P9:G1 + 3 + 4 + 5	23	3	6	2	432 (admissible)	5	786 (admissible)

bug: the arbiter does not check R when the burst starts, but behaves as if R was $\top$. This corresponds to removing the transition from S_0 to S_4 in Fig. 12, and going to S_3 instead. An execution trace is shown in Table 4. The first burst starts with $s = \top$ in Step 3. R is $\bot$, so the arbiter counts incorrectly. The erroneous output shows up in Step 7, where the arbiter starts the next burst, which is forbidden, and thus blocked by the shield. The arbiter now thinks that it has started a burst, so it keeps $s = \bot$ until R is $\top$ 4 times. In actuality, this burst start has been blocked by the shield, so the shield waits in S_0. Only after the suppressed burst is over and the components are in sync again can the next burst can start normally.

To evaluate the performance of our tool, we ran a stress test with increasingly larger sets of safety properties for the ARM AMBA bus arbiter in [7]. Table 5 summarizes the results. The first columns list the set of specification automata and the number of states, inputs, and outputs of their product automata. The next two columns list the results for shields with a fail-safe mode and the last two columns address shields without a fail-safe mode. In both cases, the table first lists the smallest number of steps under which the shield is able to recover (adversely for the properties P1–P6, cooperatively for properties P7–P9) and second the time for synthesizing a shield in seconds. For the first six properties P1–P6, a finite number k of deviations can be guaranteed, and the results for admissible shields conform with the results for k-stabilizing shields. For the last 3 experiments P7–P9 no k-stabilizing shield exists, and the results are given for admissible shields. Both methods (to create shields with and without a fail safe mode) run sufficiently fast on all properties. In these experiments, having a fail-safe mode does not give a significant speedup to justify the tradeoff of losing the ability to recover for arbitrary failure frequencies.

Table 6 Synthesis results for the LTL patterns [16]

No.	Property	b	$\|Q\|$	$\mathcal{S}$ with fail-safe mode			$\mathcal{S}$ w/o fail-safe mode		
				Time (s)	#Latches	#AIG-Gates	Time (s)	#Latches	#AIG-Gates
1	$\mathsf{G}\,\neg p$	–	2	0.01	0	0	0.01	0	0
2	$\mathsf{F}\,r \rightarrow (\neg p\,\mathsf{U}\,r)$	–	4	0.34	2	6	0.01	3	10
3	$\mathsf{G}(q \rightarrow \mathsf{G}(\neg p))$	–	3	0.34	2	6	0.01	2	8
4	$\mathsf{G}((q \wedge \neg r \wedge \mathsf{F}\,r) \rightarrow (\neg p\,\mathsf{U}\,r))$	–	4	0.34	1	9	0.01	3	15
5	$\mathsf{G}(q \wedge \neg r \rightarrow (\neg p\,\mathsf{W}\,r))$	–	3	0.01	2	14	0.02	3	19
6	$\mathsf{F}\,p$	0	3	0.34	1	1	0.01	1	1
6	$\mathsf{F}\,p$	256	259	33	18	134	39	18	106
7	$\neg r\,\mathsf{W}\,(p \wedge \neg r)$	–	3	0.05	3	11	0.01	5	27
8	$\mathsf{G}(\neg q) \vee \mathsf{F}(q \wedge \mathsf{F}\,p)$	0	3	0.04	3	11	0.01	4	19
8	$\mathsf{G}(\neg q) \vee \mathsf{F}(q \wedge \mathsf{F}\,p)$	4	7	0.04	6	79	0.02	6	54
8	$\mathsf{G}(\neg q) \vee \mathsf{F}(q \wedge \mathsf{F}\,p)$	16	19	0.03	10	162	0.05	10	89
8	$\mathsf{G}(\neg q) \vee \mathsf{F}(q \wedge \mathsf{F}\,p)$	64	67	0.37	14	349	0.58	14	114
8	$\mathsf{G}(\neg q) \vee \mathsf{F}(q \wedge \mathsf{F}\,p)$	256	259	34	18	890	38	18	150
9	$\mathsf{G}(q \wedge \neg r \rightarrow (\neg r\,\mathsf{W}\,(p \wedge \neg r)))$	–	3	0.05	2	12	0.03	5	58
10	$\mathsf{G}(q \wedge \neg r \rightarrow (\neg r\,\mathsf{U}\,(p \wedge \neg r)))$	12	14	5.4	14	2901	168	16	49840

3.9.2 LTL specification patterns

Dwyer et al. [16] studied frequently used LTL specification patterns in verification. As an exercise, we applied our tool to the first 10 properties from their list [1] and summarized the results in Table 6. For a property containing liveness aspects (e.g., something must happen eventually), we imposed a bound on the reaction time to obtain the safety (bounded-liveness) property. The bound on the reaction time is shown in Column 3. The next column lists the number of states in the safety specification. The last columns list the synthesis time in seconds, and the shield size (latches and AIG gates) for k-stabilizing shields with and without a fail-safe mode. Overall, both methods run sufficiently fast on all properties and the resulting shield size is small. For certain benchmarks we achieve a significant speedup and significantly smaller shield sizes when introducing a fail-safe mode. We also investigated how the synthesis time increased with an increasingly larger bound b. For Property 8 and Property 6, the run time and shield size remained small even for large automata. For Property 10, the run time and shield size grew faster. The reason lies in the fact that for some properties, an error by the design results in a large set of states to be monitored, while for other properties, this set stays rather small.

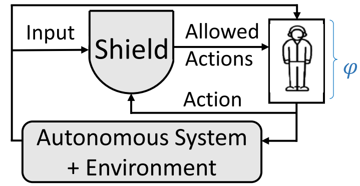


Fig. 13 Preemptive shielding

4 Shield synthesis for human–autonomy interaction

In the second part of the paper, we consider shielding a human operator who works with an autonomous system. In the context of shielding a human operator we often refer to outputs of the human operator as actions selected by the operator. Shielding a human operator instead of shielding a reactive system requires two innovations in the shielding procedure. (1) In the case of shielding reactive systems, the shield is attached *after* the system and corrects the output of the system—see Fig. 1. In contrast, when shielding human operators we attach the shield *before* the operator and the shield restricts the possible actions of the operator—see Fig. 13. We call these types of shields *preemptive shields*. (2) When shielding a human operator, it is necessary to provide simple and intuitive explanations to the operator for the interferences of the shield. We call shields that are able to provide such explanations *explanatory shields*. In the next sections, we address these two innovations in more detail.

4.1 Preemptive shields

Consider a setting in which a human operator controls an autonomous reactive system: in every time step, the environment and the system provide an input (sensor measurements, state information) to the operator, then the operator selects the next action for the system to be executed, the system executes the selected action, and the system and the environment move to the next state.

The human operator has to select actions in such a way that a given safety specification $\varphi = (Q, q_0, \Sigma_I \times \Sigma_O, \delta, F)$ with $\mathcal{A} = \Sigma_O$ is the set of available actions, is met. In any given situation an action $a \in \mathcal{A}$ is called *unsafe* if after which the system and the environment can force the specification to be violated. Otherwise, an action is called *safe*. We modify the loop between the human operator and the system, depicted in Fig. 13. The shield is implemented *before* the human operator and acts each time the operator is to make a decision by providing a list of safe actions. The task of the shield is to modify the set of available actions of the human operator in every time step such that only safe actions remain.

The interaction between the environment, the autonomous system, the human operator, and the shield is as follows: At every time step, the shield computes a set of all safe actions $\{a_1, \ldots, a^k\}$, i.e., it takes the set of all actions available, and removes all unsafe actions that would violate φ (in the worst case over future inputs). The human operator receives this list from the shield, and picks an action $a \in \{a_1, \ldots, a^k\}$ from it. The autonomous system executes action a, the system and the environment move to the next state and provide the next input to the shield and the operator.

More formally, for a preemptive shield $\mathcal{S} = (Q^S, q_0^S, \Sigma_I^S, \Sigma_O^S, \delta^S, \lambda^S)$, we have $\Sigma_O^S = 2^{\mathcal{A}}$, as the shield outputs the set of actions for the human operator to choose from for the respective next step. The shield observes the same inputs as the human operator. For selecting the next transition of the finite-state machine that represents the shield, it also makes use of

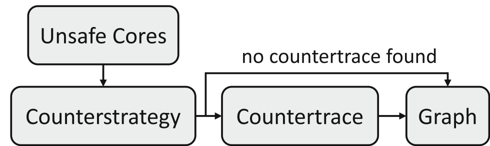


Fig. 14 Procedure to explain unsafe outputs

the action actually chosen by the human operator. So for the input alphabet of the shield, we have $\Sigma_I^S = \Sigma_I \times \mathcal{A}$.

When shielding a human operator, preemptive shielding provides many advantages. First, the shield ensures that a given safety specification is assured. Second, the shield is least restrictive to the operator, since it allows the operator to pick any action, as long as it is safe.

4.1.1 Synthesizing preemptive shields

Given is a setting in which a human operator has to control an autonomous system in an unknown environment while satisfying a safety specification $\varphi = (Q, q_0, \Sigma_I \times \Sigma_O, \delta, F)$ with $\mathcal{A} = \Sigma_O$ is the set of available actions the human operator can choose from. Starting from φ, our shield synthesis procedure is as follows:

1. Consider φ as a safety game and compute its winning region $W \subseteq F$ so that every system $\mathcal{D} \models \varphi$ must produce outputs such that the next state of φ stays in W.
2. We translate Q and W to a reactive system $\mathcal{S} = (Q^S, q_0^S, \Sigma_I^S, \Sigma_O^S, \delta^S, \lambda^S)$ that constitutes the shield with the state space $Q^S = Q$, the initial state $q_0^S = q_0$, the input $\Sigma_I^S = \Sigma_I \times \mathcal{A}$, the output $\Sigma_O^S = 2^{\mathcal{A}}$, the transition relation δ^S and the output function λ^S such that

$$\delta^S(q, \sigma_I, a) = \delta(q, \sigma_I, a) \text{ for all } q \in Q, \sigma_I \in \Sigma_I, a \in \mathcal{A} \text{ and}$$
$$\lambda^S(q, \sigma_I) = \{a \in \mathcal{A} \mid \delta(q, \sigma_I, a) \in W\} \text{ for all } q \in Q, \sigma_I \in \Sigma_I.$$

The shield allows all actions that are guaranteed to lead to a state in W, no matter what the next input is. Since these states, by the definition of the set of winning states, are exactly the ones from which the system player can enforce not to ever visit a state not in F, the shield is minimally interfering. It disables all actions that may lead to an unsafe state.

4.2 Explanatory shields

Explanatory shields provide a simple diagnosis to the operator and explain why certain actions are unsafe in the current situation. They are particulary useful in cases in which φ is very complex, e.g., consists of thousands of states, and it is difficult for the operator to comprehend why the shield had to interfere. To explain unsafe actions, we propose to use techniques for debugging formal specifications [23].

Figure 14 depicts the flow of our method to explain unsafe actions. First, we compute a minimal set of properties φ_i and signals that on their own are sufficient to show that an action is unsafe in a given situation. We call this part of the specification that contains the root causes for an action to be unsafe an *unsafe core*. Using an unsafe core, we compute a counterstrategy to explain why the action is indeed unsafe. Since counterstrategies may be large, we apply a heuristic to search for a countertrace, i.e., a single input trace which necessarily leads to a specification violation. Finally, we provide the operator with a graph that summarizes all plays that are possible against the counterstrategy (or the countertrace, in case the heuristic succeeds). Next, we will detail these steps.

4.2.1 Unsafe cores

Understanding why an action for a given state-input pair is unsafe may be difficult, but often only a small part of the specification is responsible. Removing extraneous parts from the specification gives a specification that still forbids the action but is much smaller and thus easier to understand. We call this part of the specification the *unsafe core* for a given action in a specific situation, i.e., a state-input combination.

Typically, a safety specification φ is composed of several safety properties $\varphi = \{\varphi_1 \dots \varphi_l\}$, where each φ_i defines a relatively self contained and independent aspect of the system behavior. Our goal is to identify minimal sets of properties φ_i that explain the unsafe action on their own.

Definition 10 *(Unsafe Core)* Let $\varphi = \{\varphi_1, \dots, \varphi_l\}$ with $\varphi_1 \times \dots \times \varphi_l = (Q, q_0, \Sigma_I \times \Sigma_O, \delta, F)$ be a safety specification. A subset $\phi \subseteq \varphi$ defines an *unsafe core* in state $q \in Q$ and for an input $\sigma_I \in \Sigma_I$ and an output $\sigma_O \in \Sigma_O$ if executing $\sigma_I \sigma_O$ from q results in a next state $q' \in Q$ outside the winning region of φ (when φ is interpreted as a game), and there is no strict subset of φ for which the same holds.

In the computation of unsafe cores, one can also remove signals from the specification in addition to properties. Removal of signals allows the operator to focus on those signals that are relevant for the problem at hand. It also simplifies the following counterstrategies, making them free of irrelevant signals.

Unsafe cores are very similar to unrealizable cores [12], i.e., parts of φ that are unrealizable on their own. Computing unsafe cores reduces to computing unrealizable cores, which can efficiently be computed by computing minimal hitting sets [26].

4.2.2 Counterstrategies and countertraces

Suppose we are given a safety specification $\varphi = (Q, q_0, \Sigma_I \times \Sigma_O, \delta, F)$ to be enforced by $\mathcal{S}$. Explaining why an output σ_O from a state q and an input σ_I is unsafe boils down to presenting that from $q' = \delta(q, \sigma_I, \sigma_O)$, there exists a strategy for selecting future inputs such that a state outside F is reached eventually, no matter how future outputs are selected. This strategy for selecting such inputs is called a counterstrategy. Understanding the counterstrategy implies understanding why the output is unsafe.

In general, a counterstrategy cannot be presented as a single trace of inputs, since inputs may depend on previous outputs. The dependencies can become quite complex, especially for large specifications, which makes it difficult for the operator to comprehend which environment behavior leads to unsafety. The operator may prefer one single trace of inputs such that there are no future outputs able to satisfy φ. Such a trace is called a countertrace. Unfortunately, a countertrace does not always exist. Even if one exists, its computation is often expensive. We therefore propose to use a heuristic to compute countertraces as presented in [23].

4.3 Case study on UAV mission planning

In this section, we apply shields on a scenario in which an unmanned aerial vehicle (UAV), controlled by a human operator, must maintain certain properties while performing a surveillance mission in a dynamic environment. We show how a preemptive explanatory shield can be used to enforce the desired properties and to provide feedback to the operator for any restrictions on the commands available.

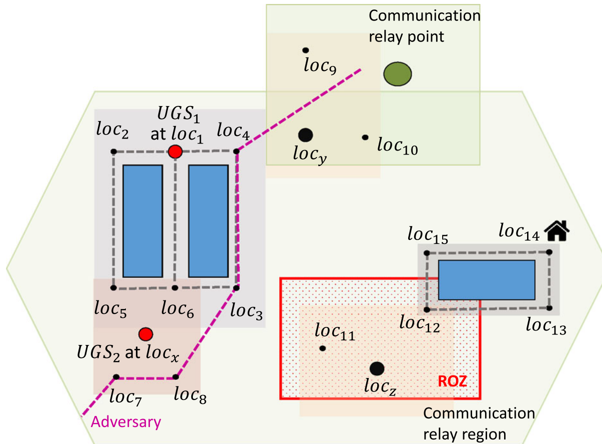


Fig. 15 A map for UAV mission planning

To begin, note that a common UAV control architecture consists of a ground control station that communicates with an autopilot onboard the UAV [11]. The ground control station receives and displays updates from the autopilot on the UAV's state, including position, heading, airspeed, battery level, and sensor imagery. It can also send commands to the UAV's autopilot, such as waypoints to fly to. A human operator can then use the ground control station to plan waypoint-based routes for the UAV, possibly making modifications during mission execution to respond to events observed through the UAV's sensors. However, mission planning and execution can be workload intensive, especially when operators are expected to control multiple UAVs simultaneously [14]. Errors can easily occur in this type of human–automation paradigm, because a human operator might neglect some of the required safety properties due to high workload, fatigue, or an incomplete understanding of exactly how a command is executed.

As the mission unfolds, waypoint commands will be sent to the autopilot. A shield that monitors the inputs and restricts the set of available waypoints would be able to ensure the satisfaction of the desired properties. Additionally, a shield should explain any restrictions it makes in a simple and intuitive way to the operator.

Consider the mission map in Fig. 15 [21], which contains three tall buildings (illustrated as blue blocks), over which a UAV should not attempt to fly. It also includes two unattended ground sensors (UGS) that provide data on possible nearby targets, one at location loc_1 and one at loc_x, as well as two locations of interest, loc_y and loc_z. The UAV can monitor loc_x, loc_y, and loc_z from several nearby vantage points. The map also contains a restricted operating zone (ROZ), illustrated with a red box, in which flight might be dangerous, and the path of a possible adversary that should be avoided (the pink dashed line). Inside the communication relay region (large green area), communication links are highly reliable. Outside this region, communication relies on relay points with lower reliability.

Given this scenario, properties of interest include:

P1. *Adjacent waypoints* The UAV is only allowed to fly to directly connected waypoints.
P2. *No communication* The UAV is not allowed to stay in a zone with reduced communication reliability and has to leave this zone within 1 time step.
P3. *Restricted operating zones* The UAV has to leave a ROZ within 2 time steps.

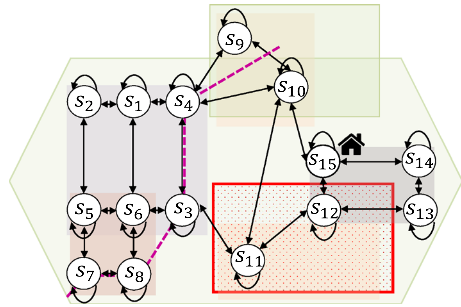


Fig. 16 Safety automaton of Property P1

P4. *Detected by an adversary* Locations on the adversary's path cannot be visited more than once over any window of 3 time steps.
P5. *UGS* (a) If UGS_1 reports a target, the UAV should visit loc_1 within 7 steps. (b) If UGS_2 reports a target, the UAV should visit loc_5, loc_6, loc_7, or loc_8 within 7 steps.
P6. *Refuel* Once the UAV's battery is low, it should return to a designated landing site at loc_{14} within 10 time steps.

The task of the shield is to ensure these properties during operation. In this setting, the human operator responds to mission-relevant inputs, e.g., in this case data from the UGSs and a signal indicating whether the battery is low. In each step, the operator sends the next waypoint to the autopilot, which is encoded in a bit representation via outputs l_4, l_3, l_2, and l_1. The shield is implemented *before* the operator. It monitors mission inputs, and provides a list of safe waypoints to the operator each time the operator is going to select a next one. This list restricts the choices of the operator.

We represent each of the properties by a safety automaton, the product of which serves as the shield specification φ. Figure 16 models the "connected waypoints" property, where each state represents a waypoint with the same number. Edges are labeled by the values of the variables $l_4 \dots l_1$. For example, the edge leading from state s_5 to state s_6 is labeled by $\neg l_4 l_3 l_2 \neg l_1$. For clarity, we drop the labels of edges in Fig. 16. The automaton also includes an error state, which is not shown. Missing edges lead to this error state, denoting forbidden situations.

For our experiments, we used the six Properties P1-P6 as safety specification $\varphi_1, \dots, \varphi_6$. We synthesized preemptive shields as described in Sect. 4.1.1. To provide explanations for unsafe actions, we computed the unsafe core of the properties $\{\varphi_1, \dots, \varphi_6\}$. All results are summarized in Table 7. The left four columns list the set of properties and the number of states, inputs, and outputs of their product automata, respectively. The last column lists the time to synthesize the explanatory preemptive shield. Note that in order to compute the unsafe cores, it is necessary to compute shields for all possible subsets of properties. The synthesis time given in Table 7 is the total time including the creation of all shields used to compute the unsafe cores.

All computation times are for a computer with an Intel Xeon 4.0GHz CPU and 16GB RAM running a 64-bit distribution of Linux.

We integrated our shields in the AMASE multi-UAV simulator [15]. AMASE is a flight simulation environment, which models UAVs using a kinematic flight dynamics model that includes environmental effects (e.g., wind) on performance. Fig. 17 visualizes the map of Fig. 15 using AMASE and shows one UAV (in blue), currently in the ROZ at location loc_{11} and controlled by the human operator, and a second adversarial UAV (in pink) on its way to loc_8. We integrate a shield to ensure the six safety properties P1–P6. On the right-hand

Table 7 Results for UAV experiments

Property	$\|Q\|$	$\|I\|$	$\|O\|$	Time (s)
1	16	0	4	0.01
1+2	16	0	4	0.13
1+2+3	19	0	4	0.27
1+2+3+4	23	0	4	0.92
1+5a	84	1	4	0.9
1+5a+2	84	1	4	3.95
1+5a+2+3	100	1	4	12.48
1+5b	64	1	4	0.5
1+5b+2	64	1	4	1.97
1+6	115	1	4	1.6

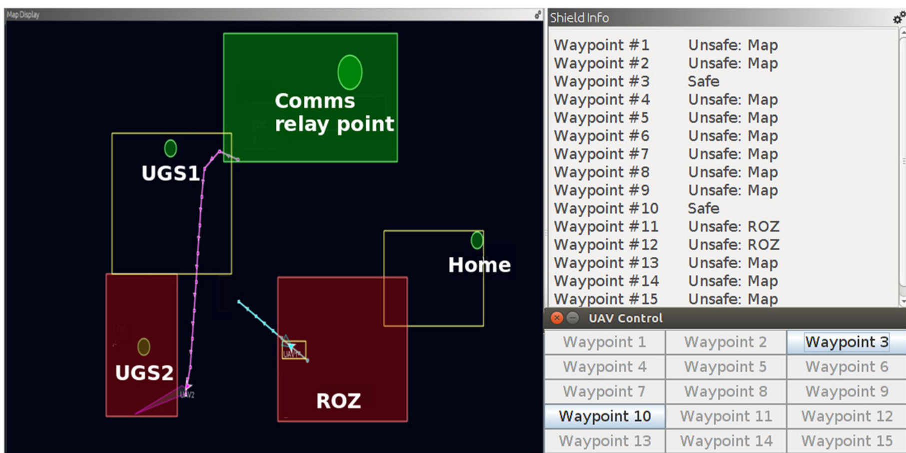


Fig. 17 Simulation of explanatory preemptive shield developed on AMASE

side of the graphical interface, the operator can select the next waypoint for the controlled UAV. In the current situation, adjacent waypoints for the controlled UAV are loc_3, loc_{10}, loc_{11}, and loc_{12}, with loc_3 and loc_{10} being outside the ROZ. The shield disables all other waypoints because the UAV is only allowed to fly to directly connected waypoints, according to Property P1. Assume that the controlled UAV already spent two time steps in the ROZ. Therefore, it has to leave the ROZ in the next time step, according to Property P3, and the shield disables the waypoints at location loc_{11} and loc_{12} as well. The explanation for any restrictions made by the shield are illustrated in the upper right corner of the GUI.

5 Related work

Our work builds on synthesis of reactive systems [7,33] and reactive mission plans [17] from formal specifications, and our method is related to synthesis of robust [4] and error-resilient [18] systems. However, our approach differs in that we do not synthesize an entire system, but rather a shield that considers only a small set of properties and corrects the

output of the system at runtime. Li et al. [25] focused on the problem of synthesizing a semi-autonomous controller that expects occasional human intervention for correct operation. A human-in-the-loop controller monitors past and current information about the system and its environment. The controller invokes the human operator only when it is necessary, but as soon as a specification will be violated, such that the human operator has sufficient time to respond. Similarly, our shields monitor the behavior of systems at run time and interfere as little as possible. Our work relates to more general work on runtime enforcement of properties [20], but shield synthesis is the first appropriative work for reactive systems, since shields act on erroneous system outputs immediately without delay.

This paper extends our previous work on shield synthesis [8,22]. In [8] we defined the general framework for solving the shield synthesis problem for reactive systems, and proposed the k-stabilizing shield synthesis procedure that guarantees recovery in k steps. Admissible shields were proposed in [22]. Here we assume that systems generally have cooperative behavior with respect to the shield, i.e., the shield ensures a finite number of deviations if the system chooses certain outputs. This is similar in concept to cooperative synthesis as considered in [6], in which a synthesized system has to satisfy a set of properties (called guarantees) only if certain environment assumptions hold. The authors present a synthesis procedure that maximizes the cooperation between system and environment for satisfying both guarantees and assumptions as much as possible. This work extends the previous work on shield synthesis by (1) discussing how liveness properties of the design can be preserved when shielded, (2) introducing a procedure for shielding the decisions of a human operator, and (3) presenting new experiments.

6 Conclusion

In this paper, we have presented automated synthesis of shields for reactive systems. Given a set of safety specifications, a shield monitors the inputs and outputs of the reactive system and corrects erroneous output values at runtime. The shield deviates from the given outputs as infrequently as it can and recovers to hand back control to the system as soon as possible. We provided theoretical and algorithmic background for three concrete instantiations of the shield concept: (1) k-stabilizing shields that guarantee recovery in a finite time. (2) Admissible shields that attempt to work with the system to recover as soon as possible. (3) An extension of k-stabilizing and admissible shields in which erroneous output values are corrected such that liveness properties of the system are preserved. The results from numerical experiments illustrate the scalability and effectiveness of the shield synthesis methods. In addition to these numerical experiments on existing benchmarks, we illustrated the broad applicability of shielding in a case study on human–autonomy interactions. We considered shielding the decisions of a human operator who works with an autonomous system. In this setting, we pursued shields that not only enforce correctness but also support the operator with intuitive explanations of the sources of potential wrong behavior and any restrictions caused by the shield. We presented results involving mission planning for unmanned aerial vehicles. We foresee a number of potential future research directions. These include shielding adaptive systems whose functionality evolves over time and systems that implement various learning algorithms. In general, we expect the conceptual simplicity of shielding to be offer a novel approach to develop artificially intelligent systems with provable safety and correctness guarantees.

Acknowledgements Open access funding provided by Austrian Science Fund (FWF). The study was funded by Austrian Science Fund (Grant No.S11406-N23).

Open Access This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

References

1. LTL Specification Patterns. http://patterns.projects.cis.ksu.edu/documentation/patterns/ltl.shtml
2. Alpern B, Schneider FB (1985) Defining liveness. Inf Process Lett 21(4):181–185
3. Bernet J, Janin D, Walukiewicz I (2002) Permissive strategies: from parity games to safety games. ITA 36(3):261–275
4. Bloem R, Chatterjee K, Greimel K, Henzinger TA, Hofferek G, Jobstmann B, Könighofer B, Könighofer R (2014) Synthesizing robust systems. Acta Inf 51(3–4):193–220
5. Bloem R, Chatterjee K, Greimel K, Henzinger TA, Jobstmann B (2010) Robustness in the presence of liveness. In: Proceedings of computer aided verification, 22nd international conference, CAV 2010, Edinburgh, 5–19 July 2010, pp 410–424
6. Bloem R, Ehlers E, Könighofer R (2015) Cooperative reactive synthesis. In: ATVA, LNCS, vol. 9364. Springer, pp 394–410. doi:10.1007/978-3-319-24953-7
7. Bloem R, Jobstmann B, Piterman N, Pnueli A, Sa'ar Y (2012) Synthesis of reactive(1) designs. J Comput Syst Sci 78(3):911–938
8. Bloem R, Könighofer B, Könighofer R, Wang C (2015) Shield synthesis: runtime enforcement for reactive systems. In: Tools and algorithms for the construction and analysis of systems—21st international conference, TACAS 2015, London, 11–18 Apr 2015, LNCS, vol. 9035. Springer, pp 533–548
9. Brayton RK, Mishchenko A (2010) ABC: an academic industrial-strength verification tool. In: CAV, LNCS, vol 6174, Springer, pp 24–40
10. Bryant RE (1986) Graph-based algorithms for boolean function manipulation. IEEE Trans Comput 35(8):677–691
11. Chao H, Cao Y, Chen Y (2010) Autopilots for small unmanned aerial vehicles: a survey. Int J Control Autom Syst 8(1):36–44
12. Cimatti A, Roveri M, Schuppan V, Tchaltsev A (2008) Diagnostic information for realizability. In: Proceedings of verification, model checking, and abstract interpretation, 9th international conference, VMCAI 2008, San Francisco, 7–9 Jan 2008, pp 52–67
13. Dalamagkidis K, Valavanis KP, Piegl LA (2011) On integrating unmanned aircraft systems into the national airspace system: issues, challenges, operational restrictions, certification, and recommendations, vol 54. Springer, Berlin
14. Donmez B, Nehme C, Cummings ML (2010) Modeling workload impact in multiple unmanned vehicle supervisory control. IEEE Trans Syst Man Cybern A Syst Hum 40(6):1180–1190
15. Duquette M (2009) Effects-level models for UAV simulation. In: AIAA modeling and simulation technologies conference
16. Dwyer MB, Avrunin GS, Corbett JC (1999) Patterns in property specifications for finite-state verification. In: ICSE, ACM, pp 411–420
17. Ehlers R, Könighofer R, Bloem R (2015) Synthesizing cooperative reactive mission plans. In: 2015 IEEE/RSJ international conference on intelligent robots and systems, IROS 2015, Hamburg, 2015, IEEE, pp 3478–3485
18. Ehlers R, Topcu U (2014) Resilience to intermittent assumption violations in reactive synthesis. In: 17th international conference on hybrid systems: computation and control, HSCC'14, Berlin, 15–17 Apr 2014, ACM, pp 203–212. doi:10.1145/2562059.2562128
19. Faella M (2009) Admissible strategies in infinite games over graphs. In: Mathematical foundations of computer science 2009, 34th international symposium, MFCS 2009, Novy Smokovec, 2009, LNCS, vol. 5734. Springer, pp 307–318
20. Falcone Y, Fernandez J, Mounier L (2012) What can you verify and enforce at runtime? STTT 14(3):349–382
21. Feng L, Wiltsche C, Humphrey L, Topcu U (2016) Synthesis of human-in-the-loop control protocols for autonomous systems. In: IEEE/RSJ international conference on intelligent robots and systems (IROS)

22. Humphrey L, Könighofer B, Könighofer R, Topcu U (2016) Synthesis of admissible shields. In: Proceedings of hardware and software: verification and testing—12th international haifa verification conference, HVC 2016, Haifa, 14–17 Nov 2016, pp 134–151
23. Könighofer R, Hofferek G, Bloem R (2013) Debugging formal specifications: a practical approach using model-based diagnosis and counterstrategies. STTT 15(5–6):563–583
24. Leucker M, Schallhart S (2009) A brief account of runtime verification. J Log Algebr Program 78(5):293–303
25. Li W, Sadigh D, Sastry S, Seshia S (2014) Synthesis for human-in-the-loop control systems. In: Proceedings of tools and algorithms for the construction and analysis of systems—20th international conference, TACAS 2014, Grenoble, 5–13 Apr 2014, LNCS, vol. 8413. Springer, pp 470–484
26. Liffiton MH, Sakallah KA (2008) Algorithms for computing minimal unsatisfiable subsets of constraints. J Autom Reason 40(1):1–33
27. Loh R, Bian Y, Roe T (2009) UAVs in civil airspace: safety requirements. IEEE Aerosp Electr Syst Mag 24(1):5–17
28. Lygeros J, Godbole DN, Sastry S (1996) Verified hybrid controllers for automated vehicles. IEEE Trans Autom Control 43:522–539
29. Mancini T, Mari F, Massini A, Melatti I, Tronci E (2014) Anytime system level verification via random exhaustive hardware in the loop simulation. In: 17th Euromicro conference on digital system design (DSD), 2014, pp 236–245
30. Mazala R (2001) Infinite games. In: Grädel E, Thomas W, Wilke T (eds) Automata, Logics, and Infinite Games: A Guide to Current Research [outcome of a Dagstuhl seminar, February 2001]. Lecture Notes in Computer Science, vol 2500, pp 23–42. Springer. doi:10.1007/3-540-36387-4_2
31. Piterman N, Pnueli A (2006) Faster solutions of Rabin and Streett games. In: 21st symposium on logic in computer science, IEEE press, pp 275–284
32. Pnueli A (1977) The temporal logic of programs. In: 18th annual symposium on foundations of computer science, Providence, 31 Oct–1 Nov 1977, pp 46–57
33. Pnueli A, Rosner R (1989) Automata, languages and programming. In: Proceedings of 16th international colloquium Stresa, 11–15 July 1989, chap. On the synthesis of an asynchronous reactive module, Springer, Berlin, Heidelberg, pp 652–671