Shield Synthesis

Runtime Enforcement for Reactive Systems

by

Bettina Könighofer

A PhD Thesis
Presented to the Faculty of Computer Science and Biomedical Engineering,
Graz University of Technology (Austria),
in Partial Fulfillment of the Requirements for the PhD Degree

Assessors

Prof. Roderick Bloem (Graz University of Technology, Austria)
Prof. Ufuk Topcu (University of Texas at Austin, USA)

December 16, 2019

Institute for Applied Information Processing and Communications (IAIK)
Faculty of Computer Science and Biomedical Engineering
Graz University of Technology, Austria

"We can only see a short distance ahead, but we can see plenty there that needs to be done."

Alan Mathison Turing

Abstract

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, energy consumption, and pollution; and unmanned systems that can safely and efficiently operate on land, underwater, in the air, and in space. However, in each of these domains, concerns about safety are being raised. 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. Runtime enforcement is a powerful technique to ensure that a running system respects some specified properties.

This dissertation presents a new general synthesis approach for runtime enforcement for reactive systems called *shield synthesis*. We specify the safety properties which have to be obeyed by the system in temporal logic. From this specification, we automatically synthesize a correct-by-construction runtime reinforcement module called a *shield*. A shield enforces safety properties of a running system while being *minimal interfering* with the system. The shield synthesis approach is based on solving a 2-player game.

A shield can be attached to a system in two alternative ways. A *post-posed shield* is implemented after the system. It monitors the system and corrects the system's output only if necessary and as little as possible. A *preemptive shield* is placed before the system and provides a list of safe outputs to the system at each time step. This list restricts the choices of the system.

This thesis proposes different types of shields to be applied in various application areas. We discuss *k-stabilizing shields* and *admissible shields* with the ability to recover from system errors as soon as possible. We define *explanatory shields*, which provide for each single deactivated output the information, which part of the specification was responsible for the deactivation. Furthermore, we propose deterministic and probabilistic shields for *reinforcement learning agents* to guarantee safety with certainty and with high probability during the learning phase and the execution phase.

The thesis discusses the strengths and weaknesses of shields and shows their potential and versatility by discussing the results of several case studies, by pointing out the impact that shields already have on the research community and beyond, and by giving several promising directions for future work.

Kurzfassung

Der technologische Fortschritt ermöglicht die Entwicklung von intelligenten Systemen mit stetig steigender Komplexität. Kleinere und schnellere Mikroprozessoren, drahtlose Vernetzung, und neue theoretische Ergebnisse in Bereichen des maschinellen Lernens und intelligenter Steuerung ebnen den Weg für transformative Technologien in einer Vielzahl von Bereichen – autonome Fahrzeuge mit dem Potential Unfälle, Verkehr, Energieverbrauch und Umweltverschmutzung zu reduzieren; und unbemannte Systeme, die sicher und effizient an Land, unter Wasser, in der Luft und im Weltraum agieren können. Aufgrund der hohen Komplexität heutiger Systeme sind herkömmliche Test- und Verifizierungsansätze nicht mehr ausreichend und alternative Ansätze wie Formale Methoden, welche nachweisliche Sicherheitsgarantien liefern können, sind erforderlich.

Diese Dissertation präsentiert einen neuen, allgemeinen Syntheseansatz, genannt *Shield Synthesis*, um Fehler von reaktiven Systemen zur Laufzeit zu korrigieren. Sicherheitskritische Eigenschaften des Systems werden in temporaler Logik spezifiziert. Basierend auf dieser Spezifikation wird automatisch durch das Lösen eines 2-Spieler-Spiels ein beweisbar richtiges Modul zur Fehlerkorrektur erzeugt, genannt ein *Shield*. Ein Shield garantiert sicherheitskritische Eigenschaften und schränkt dabei die Systemausführung so wenig wie möglich ein.

Ein Shield kann einem System vor- oder nachgeschaltet werden. Ein nachgeschaltetes Shield wird *Post-posed Shield* genannt. Ein Post-posed Shield überwacht das System und ändert seine Ausgaben nur ab, wenn dies unbedingt erforderlich ist, um die Spezifikation zu erfüllen. Ein vorgeschaltetes Shield, genannt ein Preemptive Shield, berechnet zu jedem Zeitpunkt eine Liste sicherer Ausgaben. Das System darf nur Ausgaben aus dieser Liste wählen.

In dieser Dissertation werden verschiedene Typen von Shields erörtert. *k-Stabilizing Shields* und *Admissible Shields* sind in der Lage, nach Systemfehlern die Kontrolle an das System so schnell wie möglich zurück zu geben. *Explanatory Shields* liefern zu jeder deaktivierten Ausgabe die Information, warum eine Deaktivierung notwendig war. Des Weiteren werden Shields für bestärkendes Lernen diskutiert. Wir erörtern deterministische und probabilistische Shields für Lernagenten, um die Sicherheit während der Lern- und Anwendungsphase mit Gewissheit oder mit hoher Wahrscheinlichkeit zu garantieren.

Diese Dissertation diskutiert die Stärken und Schwächen von Shields und weist auf deren Potential und Vielseitigkeit hin, indem die Ergebnisse mehrerer Fallstudien diskutiert werden, die steigende Verbreitung von Shields im akademischen Bereich und darüber hinaus aufgezeigt wird und verschiedenste vielversprechende Richtungen für zukünftige Forschung erläutert werden.

Acknowledgements

I am indebted to numerous people who supported me in my doctoral studies and this thesis. First of all, I thank my supervisor Roderick Bloem for his trust and confidence in my abilities right from the start, for sparking my interest in research, and for guiding my work with his expertise and farsightedness. I could not have hoped for a better advisor. I also thank all my colleagues from my working group for the countless discussions and fruitful collaborations. A special thanks goes to Robert Könighofer, Georg Hofferek and Ayrat Khalimov for monitoring me, especially at the beginning of my studies. I thank all my co-authors, especially Rüdiger Ehlers who reviewed my phd-proposal, Ufuk Topcu, whom I had the pleasure to visit for research collaboration at the University at Austin in Texas and who came the long way to Austria from Texas specifically for my phd-defense. Moreover, I thank all my friends, especially Ariane Wagner, Ursula Urwanisch, Georg Hofferek and Florian Lorber for motivating me to write this thesis, and additional thanks goes to Florian for babysitting, which often made my life a lot easier. I thank my parents, especially my mum, for enduring all the ups and downs, all the progress and the delays that writing a thesis with two little kids brings with it.

I thank my two amazing kids Benedikt and Nikolas for being my source of happiness. Nothing has made me prouder than to watch both of you grow up. Last but not least, I thank Robert Eberhardt for being in my life. Robert has always been there in the smallest ways and the biggest ways. I could never ask for a better partner or friend.

Funding. The research in this thesis has been financially supported by the Austrian Science Fund (FWF) through the national research network RiSE (S11406-N23) and by Graz University of Technology.

Bettina Könighofer
Graz, December 2019

Table of Contents

List of Tables

List of Figures

1 Introduction

1.1 Background and Motivation

The future of mobility is autonomous. Most major automobile manufacturers are experimenting with fully autonomous vehicles and advanced driver-assist systems are already installed as standard equipment. Designers of trains and plains are deploying partially autonomous control systems and are experimenting with fully autonomous systems. These autonomous systems are increasingly sophisticated and complex and make extensive use of machine learning, such as reinforcement learning, and other optimization techniques for smart control in open environments [Ful18]. However, due to the complexity of such systems, it is practically infeasible to cover the entire input space with test cases. Hence, for critical applications, testing alone is often not enough to achieve a satisfying level of confidence in the correctness of a system [LBR09, DVP11].

Model checking [CE81, QS82] can formally verify that a hardware or software system satisfies a temporal logic specification. It exhaustively considers all possible input scenarios for a system, thereby eliminating the incompleteness that is inherent in testing. Model checking is often used to verify systems at *design time*, but this is not always realistic. Some systems are too large to be fully verified: for complex systems, the number of inputs and states of the system can be enormous. Thus, model checking approaches suffer from scalability problems, and it may be infeasible to prove the correctness of a complex system with respect to a given specification. Other systems, 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., proprietary components or 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 non-deterministic behaviors (such as faults) as disturbances and to verify the system with respect to this disturbance model [MMM+14]. 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 [BLS11, LS09].

Reactive synthesis [Chu63, BJP+12, Pnu77] is even more ambitious than model checking since it aims to generate a provably correct reactive system from a given specification fully automatically. Reactive systems continuously interact with their environment over inputs and outputs in a synchronous way. In every time step, the environment first provides input values, after which the system responds with output values. A temporal logic specification for a reactive system defines the allowed interaction between the environment and the system. The specification may only express *what* the system shall do but does not define *how* this is to be achieved. Therefore, writing a specification can be significantly easier than implementing it. The correct-by-construction property of synthesis eliminates the need for testing, verification, and debugging of the implementation, thereby saving development time and costs.

However, in order to synthesise complete systems from given temporal logic specifications, reactive synthesis requires a complete specification, which describes every aspect of the desired system. For complex systems, writing such a specification can sometimes be as hard as implementing the system itself, or even unmanageable, because certain aspects of a system are often easier to define imperatively. Furthermore, the problem of synthesizing reactive systems from Linear Temporal Logic (LTL) [Pnu77] specifications has a doubly exponential worst-case complexity [Ros92]. Hence, synthesis algorithms may not be scalable enough to realize large specifications. *Applicability* and *scalability* are the two main reasons why synthesis of complete systems from declarative specifications is often unrealistic in practice.

Runtime enforcement extends runtime verification by not only detecting property violations, but also altering the behavior of the system in a way that maintains the desired property [Fal10, FFM12]. Runtime enforcement has been studied in domains such as action systems: a monitor watches the current execution sequence of a system and either halts the system, or suppresses, inserts, or buffers actions, whenever the system deviates from the specified property [LBW09]. Runtime enforcement for reactive systems poses unique challenges where runtime enforcers must act without delay. In particular, when correcting reactive systems, a runtime enforcer cannot insert or delete time steps, and cannot halt the system in the case of a violation.

Synthesis of runtime reinforcement modules for reactive systems from temporal logic specifications was not studied before the work this thesis is based on.

1.2 Problem Description and Contribution

Shield Synthesis

In this thesis, we discuss a new general synthesis approach for runtime enforcement for reactive systems called *shield synthesis*. Given the temporal logic specification that is to be obeyed by a system, we propose to synthesize a reactive system called a *shield* via correct-by-construction reactive synthesis algorithms. The shield is attached to the system to enforce the specified properties at *run time*. The shield continuously monitors the input/output of the system and interferes with the system only if necessary, and as little as possible, so other non-critical properties are likely to be retained.

We propose shield synthesis as a way to complement model checking and reactive synthesis. Our goal is to enforce safety properties at runtime, even if these properties may be violated by the system. Imagine a complex system and a set of properties that cannot be proven due to scalability issues or other reasons (e.g., third-party IP cores). In this setting, we are in good faith that the properties hold, but we need to have certainty. In such situations, we propose to automatically synthesize a shield, attach the shield to the system, and the shielded system is then guaranteed to satisfy the specified properties. The shield monitors the input/output of the system and is able to correct or to restrict the system's output, but only if necessary and as little as possible.

Shields may also be used to simplify *certification*. Instead of certifying a complex system against critical requirements, we can synthesize a shield to enforce them, regardless of the behavior of the system. Then, we only need to certify this shield, or the synthesis procedure, against the critical requirements.

Advantages of shielding. Shield synthesis is a promising new direction for synthesis in general because it uses the strengths of reactive synthesis (provable correctness) while avoiding its weaknesses (scalability, applicability). The main advantages of shielding are the following:

- **Provable correctness.** The main task of a shield is to ensure correctness against a formal specification. Shields are constructed via *correct-by-construction* reactive synthesis methods that provide mathematical guarantees on correct behavior with respect to the specification. Therefore, it is guaranteed that the shielded system satisfies the safety specification.
- **Preserving optimality.** A shield is *minimal interfering* and restricts the system as little as possible and intervenes only if safe system behavior would be endangered otherwise. The minimum interference property of the

shield 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.

- **Scalability and applicability.** Shield synthesis can succeed where model checking and reactive synthesis fail because it only considers the safety critical properties, as opposed to the complex system (the system is treated as black box and it is not required to synthesize a shield), or the complete specification in the case of reactive synthesis. The set of safety-critical properties can be small and relatively easy to specify, which implies both usability and scalability.
- **Universality.** Shields can be attached to almost any system that is in constant interaction with its environment, e.g., to reactive controllers, to machine learning agents, or shields may even be used to support human operators.

Example 1
Consider the example of a path planner for autonomous vehicles. Many requirements on system behaviors such as safety concerns may be known and expressed as specifications in temporal logic and can be enforced by reactive controllers. This includes always driving in the correct lane, never jumping a red light, and never exceeding the speed limit [WET15].

The complete specification of the path planner may incorporate more subtle considerations, such as specific intentions for the current scenario and personal preferences of the human driver, such as reaching some goal quickly but at the same time driving smoothly. A path planner implementing the complete specification and optimizing all performance properties might be very complex and might be obtained by applying a machine learning algorithm, rendering complete testing or formal verification of the planner with respect to the safety properties infeasible.

A shield attached to the path planner can provide correctness guarantees concerning the safety specification while preserving the optimality provided by the path planner. Even though the path planner is very complex, shielding is manageable, since only the safety properties are considered for the construction of the shield, i.e., the correctness guarantees are agnostic to the actual path planner. Optimality is preserved as long as the planner acts correctly since the shield intervenes with the path planner only if this is necessary to guarantee safety.

Types of shields. A shield can be attached to a system in two alternative ways, depending on the location at which the shield is implemented. We distinguish between *post-posed shields* and *preemptive shields*.

- **Post-posed shields.** In post-posed shielding, depicted in Figure 1.1, the shield is introduced *after* the system. The post-posed shield monitors the inputs and outputs from the system and corrects the system's output if necessary. Thereby, the shield ensures (1) *correctness* (it corrects any

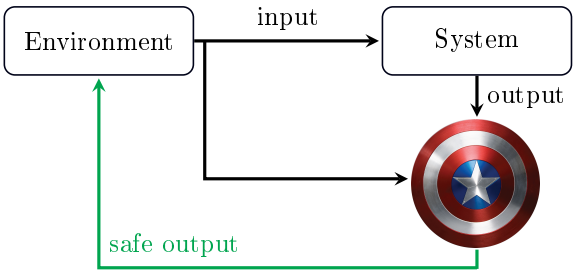


Figure 1.1: Post-posed shielding

erroneous output values instantaneously such that the safety specification is satisfied) and (2) *minimum interference* (it deviates from the system's output as little as possible). One big advantage of post-posed shielding is that the system can be designed as usually. The fact that the system will be shielded at runtime causes no additional effort in the design process of the system. During shielding, the system is used purely as a black box.

- **Preemptive shields.** In preemptive shielding, depicted in Figure 1.2, the shield is placed *before* the system and provides a list of safe outputs to the system at each time step. This list restricts the choices for the system. If the system chooses only outputs from the list of outputs provided by the shield at any time step, the preemptive shield provides (1) *correctness* (at any time step, the system can only choose safe outputs) and (2) *minimum interference* (the shield allows the system to pick any output as long as it is safe).

The concept of shielding has a broad area of application. Shields can be attached to many different types of systems to ensure safety, e.g., to reactive controllers, to human operators, or reinforcement learning agents. Depending on the setting, a shield may need to correct, restrict, or communicate with the system in particular ways. Therefore, we further categorize shields based on their properties they satisfy additionally to the two basic properties of shields (correctness and minimum interference). In this thesis, we discuss the following types of shields in detail:

- **k-Stabilizing and admissible shields.** In post-posed shielding, a shield has to correct erroneous output values instantaneously. Since the system satisfies many properties that are not considered by the shield, shields should end phases of deviations as soon as possible, recovering quickly.

  The difficulty lies in the fact, that a shield with the ability to recover has to overwrite the outputs in such a way, that the recovery phase ends as soon as possible, without any knowledge of the implementation of the system or future inputs from the environment.

  To this end, we will discuss two concrete shield synthesis methods to automatically construct shields that can recover. The resulting shields

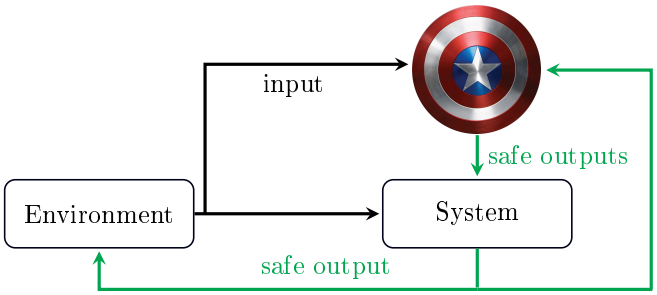


Figure 1.2: Preemptive shielding

are called *k-stabilizing shields* and *admissible shields*. k-stabilizing shields guarantee recovery in a finite time. When the system arrives at a state where a property violation becomes unavoidable for some possible future inputs, the shield is allowed to deviate for at most k consecutive steps. Whereas k-stabilizing shields take an adversarial view on the system, admissible shields take a collaborative view. That is, if there is no shield that guarantees recovery within k steps regardless of system behavior, the admissible shield will attempt to work with the system to recover as soon as possible.

- **Explanatory shields.** If the outputs to be corrected are not initiated by a system but by a human operator, it is essential to provide simple and intuitive explanations to the operator for any restrictions of the shield. When shielding human operators, we attach the shield before the operator, i.e., the shield is a preemptive shield and acts each time the operator is about to make a decision and provides a list of safe outputs. Additionally, the shield offers for each single deactivated output the information, which part of the specification was responsible for the deactivation. We call such shields *explanatory shields*.

- **Deterministic shields for reinforcement learning.** We consider post-posed and preemptive deterministic shields for learning agents and modify the loop between the learning agent and its environment in two alternative ways, respectively. In the post-posed implementation of the shield, the shield monitors the actions selected by the learning agent and corrects them if and only if the chosen action is unsafe. In the case of preemptive shielding, the shield is implemented before the learning agent and acts each time the learning agent is to make a decision and provides a list of safe actions. The shield allows the agent to follow any policy as long as it is safe according to a given safety specification. In both cases, the shield ensures safety with certainty during learning and execution.

- **Probabilistic shields for reinforcement learning.** Using a deterministic shield, a learning agent avoids safety violations altogether. However,

in many cases this tight restriction limits the agent's exploration and understanding of the environment, and policies satisfying the restrictions may not even exist.

Without randomness, all states are either absolutely safe or unsafe. However, in the presence of randomness, safety may be seen as a quantitative measure: in some states, all actions may induce a considerable risk, while in other states, one action may be considered relatively safe. A *probabilistic shield* enables decision-making to adhere to safety constraints with high probability. If an action increases the probability of a safety violation by more than a factor $1/\delta$ with respect to the optimal safety probability, the shield blocks the action from the agent. The shield is adaptive with respect to δ, as a high value for δ yields a stricter shield, and a smaller value yields a more permissive shield.

Limitations of shielding. Besides the many strengths of shields synthesis, shields also have several limitations. These limitations suggest directions for future research to overcome or to compensate them.

- **Representation of strategies.** Post-posed shields implement (memoryless) deterministic strategies, i.e., given the current state and current input, the strategy defines a fixed output. Preemptive shields implement (memoryless) non-deterministic strategies that permit per configuration many outputs instead of prescribing one. If such strategies, especially non-deterministic strategies, are computed as a list, these lists can easily take several gigabytes and cannot be exported easily. Therefore, compact representations of strategies, which can yield to acceptably short and simple code, are needed. This is still an ongoing research topic [EKH12, AKL+19].

- **Inaccurate environment models.** The computation of shields is based on a faithful abstraction of the physical environment dynamics. In case of an inaccurate model, the current approach fails, even for minor errors. Fur future work, we propose a self-correcting modeling approach using automata learning and model repair, where the model will correct itself whenever discrepancies between the model and the real environment dynamics are detected.

- **Expressiveness of specifications.** At the moment, we consider as specifications LTL safety formulas over Boolean signals. These specifications may not be expressive enough to formalize the required safety properties of cyber-physical systems, which involve real-valued signals and work in continuous time steps. To this end, as future work, we will explore more expressive specifications like specifications given in signal temporal logic (STL) of timed automata to construct shields, and we will investigate general synthesis methods for these kinds of specifications, building on the work of Wu et al. [WWDW19].

- **Complexity of k-stabilizing and admissible shields.** k-stabilizing and admissible shields use a subset construction to construct a belief set that captures all possibilities to avoid unjust verdicts by the shield. This subset construction leads to an exponential blow-up of the state space. Therefore, these types of shields can currently only be used if the properties to be shielded are relatively simple. For future work, we will apply heuristics to avoid this blow-up for concrete application scenarios [ET15].

Contributions of this thesis. In summary, the main contributions of this thesis are the following:

Contribution

- We proposed a new direction in reactive synthesis for synthesizing runtime enforcement modules, called shields. Shields guarantee correctness of the shielded system with respect to a temporal logic specification and provide minimal interference to preserve the optimality of the system.
- We proposed two basic types of shielding: preemptive and post-posed shielding.
- We enhanced post-posed shields with the ability to recover quickly and called these shields k-stabilizing and admissible shields.
- We proposed explanatory shields designed for shielding human operators that provide explanations for any actions deactivated by the shield.
- We were the first to combine reactive synthesis with reinforcement learning. We discuss deterministic and probabilistic shields to guarantee safety with certainty and with high probability during exploration and exploitation.
- For all types of shields, we discussed the setting, motivated the problem and technical challenges with examples, defined the shield, and provided synthesis algorithms to construct such shields automatically.

1.3 Related Work

Reactive synthesis. In 1962, Alonzo Church [Chu63] first posed the synthesis problem for specifications given in the so-called Monadic Second Order Logic of One Successor (S1S). A few years later, Büchi and Landweber [BL69] gave a solution to Church's problem. Unfortunately, the worst-case complexity of the synthesis problem from S1S specifications is non-elementary, i.e., cannot be expressed with a fixed number of exponentiations. In 1989, Pnueli and Rosner [PR89] studied the synthesis of reactive systems from Linear Temporal Logic (LTL) [Pnu77] specifications and established a doubly exponential lower bound for this problem. Due to these high complexities, synthesis was deemed intractable for any industrial examples and was mainly of academic interest for quite some time.

Despite these high worst-case complexities, much research effort has been invested over the last two decades on reactive synthesis theory, algorithms, and tools and the developments made synthesis techniques applicable to real-world problems. One approach is to limit the expressiveness of the specification language. For instance, Bloem et al. [BGJ+07] have shown that Generalized Reactivity of Rank 1 (GR(1)) is a specification language that is expressive enough for many applications, and that can effectively be used to synthesize examples of industrial scale. Furthermore, symbolic synthesis algorithms and tools implementing them were developed [ER16]. Existing synthesis approaches mostly use Binary Decision Diagrams (BDDs) [Bry86] as a reasoning engine. Satisfiability-based bounded synthesis [SF07] by Finkbeiner and Schewe is an alternative to classical BDD-based synthesis approaches. This approach sets a bound on the size of the systems to construct and increases this bound iteratively until an implementation is found. The rationale behind this approach is that real-world specifications typically have relatively simple implementations. Over the past decade, reactive synthesis from temporal logic specifications became a hot topic with tons of publications. To give just a few examples, consider [BJP+12, JGWB07, FJR09, SS13, JB12].

Our work on shield synthesis builds on the above-mentioned work on synthesis of reactive hardware modules from temporal logic formulas. 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.

Methodologically, our shield synthesis algorithms build upon the existing work on synthesis of robust systems [BCG+14], which aims to generate a complete system that satisfies as many properties of a specification as possible if assumptions are violated. However, our goal is to synthesize a shield component, which can be attached to any system, to ensure that the shielded system satisfies a given set of critical properties. Our k-stabilizing synthesis method aims at minimizing the ratio between shield deviations and property violations by the system, but achieves it by solving pure safety games. Furthermore, the synthesis method in [BCG+14] uses heuristics and user input to decide from which state to continue monitoring the environmental behavior, whereas we use

a subset construction to capture all possibilities to avoid unjust verdicts by the shield. We use the notion of k-stabilization to quantify the shield deviation from the system, which has similarities to Ehlers and Topcu's notion of k-resilience in robust synthesis [ET14] for GR(1) specifications [BJP+12]. However, the context of our work is different, and our k-stabilization limits the length of the recovery period instead of tolerating bursts of up to k glitches.

Runtime enforcement. Runtime verification is a technique that monitors the execution of an underlying system against a set of specified properties at runtime. Runtime enforcement is an extension of runtime verification, aiming to enforce specified behaviors. Within this technique, an enforcer not only observes the current system execution, but the enforcer also modifies it such that the execution is always correct.

In 2000, the concept of runtime enforcement was first introduced by Schneider [Sch00]. In his work, he proposed security automata, which watch the current execution sequence of a system and simply halt the system whenever it deviates from the specified property. Such security automata are able to enforce safety properties stating that something bad can never happen.

More recently, Ligatti et al. [LBW09] proposed more powerful enforcement modules called edit-automata, able to suppress or insert actions. Falcone et al. [FFM12] proposed to buffer actions and dump them once the execution is shown to be safe.

None of these approaches is appropriate for reactive systems where the enforcer must act upon erroneous outputs on-the-fly, i.e., without delay and without knowing what future inputs/outputs are. In particular, our shield cannot insert or delete time steps, and cannot halt in the case of a violation.

Li et al. [LSSS14] 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 is violated ahead of time, 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.

The term *runtime assurance* is often used in frameworks in which a switching mechanism alternates between running a high-performance system and a provably safe one [Sha01], and has applications, for example, in control of robotics [PYG+17] and drones [DGS+19].

Safe reinforcement learning. In reinforcement learning (RL), an agent acts to optimize a long-term return that models the desired behavior for the agent and is revealed to it incrementally in a reward signal as it interacts with its environment [SB98].

During the exploration, the current policy of the learning agent may be unsafe in the sense that it harms the agent or the environment. This shortcoming restricts the application of RL mainly to academic or uncritical application areas

where safety is not a concern and has triggered the direction of safe exploration for RL, in short, *safe RL* [GF15, PS14].

An exploration process is called *safe* if no undesirable states are ever visited, which can only be achieved through the incorporation of external knowledge [GF15, MA12]. The safety fragment of temporal logic that we consider is more general than the notion of safety of [GF15] (which is technically a so-called *invariance property* [BK08]). One way of guiding exploration in learning is to provide *teacher advice*. A teacher (usually a human) provides advice (e.g., safe actions) when either the learner [PS14, Clo97] or the teacher [TB06] considers it to be necessary to prevent catastrophic situations. Most approaches to safe RL [GF15, PS14] rely on reward engineering and effectively changing the learning objective. For example, in the teacher setting, the human teacher tunes the reward signal before sending it to the agent [TB06, TB08]. In contrast to ensuring temporal logic constraints, reward engineering designs, or "tweaks" the reward functions such that a learning agent behaves in a desired, potentially safe manner. As rewards are specialized for particular environments, reward engineering runs the risk of triggering negative side effects or hiding potential bugs [SPE+14]. Recently, it was shown that reward engineering is not sufficient to capture temporal logic constraints in general [HPS+19].

Our work is closely related to teacher-guided RL, since a shield can be considered as a teacher, who provides safe actions and restricts the agent only if absolutely necessary. In contrast to previous work, the reward signal does not have to be manipulated by the shield, since the shield corrects unsafe actions in the learning and exploitation phases.

Formal methods and reinforcement learning. All of the above approaches for reactive synthesis have in common that a faithful, yet precise enough, abstraction of the physical environment is required, i.e., the synthesized controllers guaranty to satisfy the specification under the known environment dynamics. Therefore, it is necessary to combine reactive synthesis with faithful environment modeling and abstraction. Wongpiromsarn et al. [WTM12] define a receding horizon control approach that combines continuous control with discrete correctness guarantees. For simple system dynamics, the controller can be computed directly [HK99]. For more complex dynamics, the approach is computationally too difficult. An adequate abstraction of the environment is not only difficult to obtain in practice but also introduces the mentioned computational burden. Control methods based on reinforcement learning partly address this problem, but do not typically incorporate any correctness guarantees.

First approaches directly incorporating formal specifications tackle this problem with pre-computations; making assumptions on the available information about the environment [WET15, JND+16, FP18, HAK18, MCKB17, MA12]. For example, Wen et al. [WET15] propose a method to combine strict correctness guarantees with reinforcement learning. They start with a non-deterministic correct-by-construction strategy and then perform reinforcement learning to limit it towards cost optimality without having to know the cost function a

priori. Junges et al. [JND+16] adopt a similar framework in a stochastic setting. A major difference between the works by Wen et al. and Junges et al. [WET15, JND+16] on the one hand and the shielding framework on the other hand is the fact that the computational cost of the construction of the shield depends on the complexity of the specification and a very abstract version of the environment, and is independent of the state space components of the environment to be controlled that are irrelevant for enforcing the safety specification.

In this thesis, we will discuss deterministic shields that guarantee safety with certainty, and probabilistic shields that guarantee safety with high probability. Consideration of stochastic behavior is natural to RL. Intuitively, without stochasticities, a learning agent does not take any risk, which is unrealistic in most scenarios. Moreover, often one cannot assume that almost-sure safety is realizable. A similar approach to our probabilistic shields was developed independently in [BKN+19], but targets a different case study and does not consider scalability issues of formal verification.

In a related direction, methods from reinforcement learning have been successfully employed to improve the scalability of verification methods for MDPs. Such approaches often use rich specifications like ω-regular languages as a control to guide the exploration of MDP during learning [SKC+14, BCC+14, HAK18, KPR18].

Safe model-based RL for continuous state spaces employing Lyapunov functions is considered in [BTSK17, CNDGG18]. Ohnishi et al. [OWNE19] use control barrier functions (CBFs) for safe RL. The tool `UPPAAL STRATEGO` provides several algorithms combining safety synthesis with optimizing RL for continuous space MDPs [DJL+15].

Probabilistic planning considers similar problems as probabilistic model checking [SHB16, Kol12]. A recent comparison between tools from both areas can be found in [HHH+19].

Follow-up shield synthesis papers. Shield synthesis opened a new research area in reactive synthesis, and especially the combination of shields with reinforcement learning agents became a hot topic in the last two years. This is witnessed by the fact that the papers on shield synthesis by the author are already cited over a hundred times, and several follow up papers were already published by the time this thesis was written. Several research groups adapted the name *shield synthesis* for their synthesis approaches. We cite a few papers that build on the papers this thesis is built on.

- M. Wu J. Wang, J Deshmukh, C. Wang: *Shield Synthesis for Real: Enforcing Safety in Cyber-Physical Systems.* arXiv. 2019 [WWDW19]
  Wu et al. extend shields for boolean signals to real-valued shields to enforce the safety of cyber-physical systems. Boolean shields do not handle real-valued signals ubiquitous in cyber-physical systems, meaning their corrections may be either unrealizable or inefficient to compute in the real domain. The proposed approach analyzes the compatibility of predicates

defined over real-valued signals, and uses the analysis result to constrain the corrections of the shield.

- H. Zhu, Z. Xiong, S. Magil, S. Jagannathan: *An Inductive Synthesis Framework for Verifiable Reinforcement Learning.* PLDI. 2019 [ZXMJ19]
  The authors developed a counterexample-guided inductive synthesis framework (CEGIS) that treats the neural control policy as an oracle to guide the search for a simpler deterministic program that approximates the behavior of the network but which is more amenable for verification. Verification methods are used to guarantee that all actions proposed by the synthesized program are safe according to the specification. The synthesized program is treated as a safety shield, overriding the proposed actions of the network whenever such actions can cause the system to enter into an unsafe region.

- O. Bastani: *Safe Reinforcement Learning via Online Shielding.* arXiv. 2019 [Bas19]
  Bastani proposed an approach based on shielding, which uses a backup controller to override the learned controller if necessary to ensure that safety holds. Rather than to compute when to use the backup controller ahead-of-time, this computation is performed online, which enables adaptivity to new environments.

- S. Bharadwaj, S. Carr, N. Neogi, H. Poonawala, A. B. Chueca, U. Topcu: *Traffic Management for Urban Air Mobility.* NFM. 2019 [BCN+19]
  The authors presented a localized shield synthesis method to generate shields to perform traffic management for an urban air mobility (UAM) systems. Their method exploits the geographic separation of the traffic management problem to avoid the full distributed synthesis problem and focus on localized shield synthesis where shields can observe the outputs of neighbors. Assume-guarantee contracts between neighboring shields are used to ensure that each shield can still satisfy its safety requirements without violating the ability of neighbors to satisfy theirs.

The following two publications on shield synthesis are from the author of this thesis but are not captured in this thesis.

- S. Bharadwaj, R. Bloem, R. Dimitrova, B. Könighofer, U. Topcu: *Synthesis of Minimum-Cost Shields for Multi-agent Systems.* ACC. 2019 [BBD+19]
  In this paper, we presented a technique for synthesizing quantitative shields for multi-agent systems. Each agent of the multi-agent system is monitored, and if needed corrected, by the shield, such that a global specification is satisfied.

  The distributed nature of the problem gives rise to a number of considerations to be made during the shield synthesis procedure. To explore the design space of possible shields for multi-agent systems, we categorized shields based on three criteria according to (1) the interference of the shield processes with the individual agents, (2) the assumptions on the

behavior of the agents the shield can rely on, and (3) the fairness of the shield with respect to the individual agents.

1. Quantifying interference. By construction, a shield is guaranteed to enforce the correct operation of the shielded system. However, we might prefer one shield over another, based on how much the shield interferes with the system as a whole, or how it interferes with the individual agents in case of an error. In this paper, we introduced the notion of *interference cost* to quantify the quality of a shield and synthesize cost-optimal shields that minimize the interference cost for the worst-case behavior of the multi-agent system. We discussed different cost functions and provided algorithms to synthesize cost-optimal shields.

2. Assumptions on the multi-agent system. The shield synthesis procedure does not rely on the particular implementation of the system or specifications of each of the agents, which is the key to the practicability of the approach. Instead, a shield has to guarantee safety for any possible implementation. However, it is often realistic to make assumptions on the worst-case behavior of the system and synthesize optimal shields with respect to the chosen interference cost under these assumptions. A natural assumption is that wrong outputs rarely occur, i.e., the length of all sequences of wrong outputs is bounded. When such knowledge is available, we computed a cost-optimal shield considering the worst-case behavior of any system satisfying the assumptions.

3. Fair shielding. In the multi-agent setting, in which each agent might have to fulfill some individual goals, it is often important that a shield treats all agents fairly: in case of an error, a fair shield does not always interfere with the same agent repeatedly. In the paper, we defined a fairness notion for shields, and discussed the corresponding synthesis procedure.

We demonstrated the applicability of our approach via a detailed case study on UAV mission planning for warehouse logistics and simulating the shielded multi-agent system on ROS/Gazebo.

- G. Avni, R. Bloem, K. Chatterjee, T. A. Henzinger, B. Könighofer, S. Pranger: *Run-Time Optimization for Learned Controllers Through Quantitative Games.* CAV. 2019 [ABC+19]
  Besides safety issues, learned controllers have further shortcomings. In particular, it is difficult to add features without retraining, to guarantee any level of performance, and to achieve acceptable performance when encountering untrained scenarios.

  In this work, we used *quantitative shields* to deal with the limitations of learned or any other black-box controllers by optimizing quantitative measures. We were interested in synthesizing lightweight shields. We assumed that the controller performs well on average, but has no worst-case guarantees. When combining the shield and the controller, intuitively, the controller should be active for the majority of the time, and the shield intervenes only when it is required. We formalized the *controller performance*

as well as the *interference cost* using quantitative measures. Unlike safety objectives, where it is clear when a shield must interfere, with quantitative objectives, a non-interference typically does not have a devastating effect. It is thus challenging to decide, at each time point, whether the shield should interfere or not; the shield needs to balance the cost of interfering with the decrease in performance of not interfering.

The problem of constructing a shield that guarantees maximal performance with minimal interference is the problem of finding an optimal strategy in a stochastic 2-player game with a quantitative objective obtained from combining the performance and interference measures. Our construction of the game can be seen as a two-step procedure: we construct a stochastic game with two mean-payoff objectives, where the shield player's goal is to minimize both the behavioral and interference scores separately. We then reduce the game to a "one-dimension" game by weighing the scores with the parameter λ.

Finally, in our setting, the parameter λ provides a meaningful tradeoff: it can be associated with how well we value the quality of the controller. If the controller is of poor quality, then we charge the shield less for interference and set λ to be low. On the other hand, for a high-quality controller, we charge the shield more for interferences and set a high value for λ.

We illustrated the effectiveness of our approach by automatically constructing lightweight shields for learned traffic-light controllers in various road networks. The shields we generated avoid liveness bugs, improved controller performance in untrained and changing traffic situations, and added features to learned controllers, such as giving priority to emergency vehicles.

1.4 Outline of this Thesis

After this introductory chapter, Chapter 2 will revisit the theoretical background on which our research is based. We will also give essential definitions and establish notation conventions.

Chapter 3 is dedicated to the motivation and the definition of shields. We will define and synthesize preemptive and post-posed shields based on their two basic properties: correctness and minimum interference.

In Chapter 4, we will discuss two concrete shield synthesis approaches to synthesize (1) k-stabilizing shields that guarantee recovery within k steps, and (2) admissible shields that recover with the help of the system. Following, we will extend the k-stabilizing shield synthesis procedure to be liveness preserving, i.e., erroneous output values of the reactive system are corrected in such a way that liveness properties of the system are preserved. To conclude this chapter, we will provide experimental results for both shield synthesis approaches.

In Chapter 5, we will discuss preemptive shields for human operators who work with autonomous systems. In this setting, we discuss explanatory shields

that aim to provide simple explanations to human operators for any interferences. We will motivate the need for shielding a human operator via a case study involving mission planning for an unmanned aerial vehicle (UAV), and we will conclude this chapter with experimental results on this case study.

In Chapter 6, we will introduce a deterministic shield in the classical reinforcement learning setting to guarantee safety with certainty during the exploration and exploitation phase, and discuss post-posed and preemptive shielded learning. Furthermore, we will discuss the consequences of shielding for convergence guarantees of the underlying learning algorithm. Finally, we will demonstrate the versatility of our approach on several challenging reinforcement learning scenarios.

As in Chapter 6, Chapter 7 is also dedicated to shielding a reinforcement learning agent, but this time, the shield exploits the underlying stochastic behavior of the MDP model of the environment. We will synthesize a probabilistic shield that forces decision-making to provably adhere to the safety requirements with high probability. This allows the learning agent to take some risks in order to foster progress in sufficiently exploring the environment. We will show tradeoffs between sufficient progress in the exploration of the environment and ensuring strict safety. In our experiments, we will show that the learning efficiency increases as the learning needs orders of magnitude fewer episodes.

Chapter 8 will summarize our work, point out the most important contributions, and talk about potential future work.

Declaration of Sources

Chapter 1 was based on and reuses material from the following sources, previously published by the author:

- [BKKW15] R. Bloem, B. Könighofer, R. Könighofer, C. Wang: *Shield synthesis - runtime enforcement for reactive systems.* TACAS. 2015
- [HKKT16] L. R. Humphrey, B. Könighofer, R. Könighofer, U. Topcu: *Synthesis of admissible shields.* HVC. 2016
- [KAB+17] B. Könighofer, M. Alshiekh, R. Bloem, L. R. Humphrey, R. Könighofer, U. Topcu, C. Wang: *Shield synthesis.* FMSD. 2017
- [ABE+18] M. Alshiekh, R. Bloem, R. Ehlers, B. Könighofer, Scott Niekum, and Ufuk Topcu: *Safe reinforcement learning via shielding.* AAAI. 2018
- [JKJB18a] N. Jansen, B. Könighofer, S. Junges, R. Bloem: *Shielded decision-making in MDPs.* ArXiv. 2018

References to these sources are not always made explicit.

2 Preliminaries

2.1 Motivation and Outline

In this section, we will revisit some theoretical background on which our work builds. We assume that the reader is already familiar with temporal logic, game theory, reactive synthesis, and reinforcement learning. Thus, we will only briefly recapitulate some important definitions to avoid ambiguities and establish a consistent notation.

Outline. In the remainder of this chapter, we will first establish some basic notation (Section 2.2), followed by the formalization of a reactive system as Mealy machine (Section 2.3). Next, we will establish some basic definitions of automata theory (Section 2.4), and we will define several basic acceptance conditions (Section 2.5). In Section 2.6, we will discuss formal specifications of reactive systems given in temporal logic. We will elaborate on game theory in Section 2.7, and we will discuss the automated synthesis of reactive systems via game solving in Section 2.8. After that, we will consider the probabilistic setting, and we will discuss Markov decision processes (Section 2.9) and reinforcement learning (Section 2.10).

2.2 Basic Notation

In general, we will use upper case letters for sets, lower case letters for set elements, and calligraphic fonts for tuples defining more complex structures. We will write *iff* as a shorthand for "if and only if".

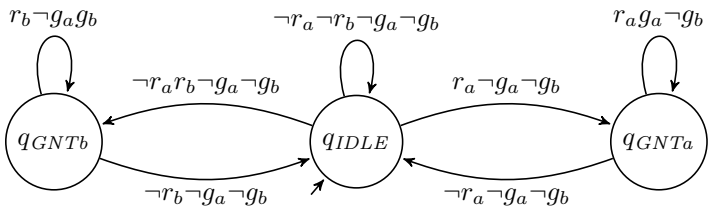


Figure 2.1: Reactive system implementing a simple arbiter.

We denote the Boolean domain by $\mathbb{B} = \{\top, \bot\}$, the set of natural numbers (including 0) by $\mathbb{N}$, and abbreviate $\mathbb{N} \cup \{\infty\}$ by $\mathbb{N}^\infty$.

A *word* is defined to be a finite or infinite sequence of elements from some alphabet Σ. The set of finite words over Σ is denoted by Σ^*, and the set of infinite words over Σ is written as Σ^ω. The union of Σ^* and Σ^ω is denoted by the symbol Σ^∞. We will also refer to words as *traces* $\overline{\sigma}$. We write $|\overline{\sigma}|$ for the length of a trace $\overline{\sigma} \in \Sigma^\infty$. A set $L \subseteq \Sigma^\infty$ of words is called a *language*. We denote the set of all languages as $\mathcal{L} = 2^{\Sigma^\infty}$.

2.3 Reactive Systems

We formalize a general model of a finite-state reactive system by a *Mealy* machine with 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 alphabet $\Sigma = \Sigma_I \times \Sigma_O$ of a reactive system composes of the input alphabet $\Sigma_I = 2^I$ and the output alphabet $\Sigma_O = 2^O$. 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 $(x_0, y_0)(x_1, y_1) \ldots \in \Sigma^\infty$.

A *Mealy machine* 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 traces produced by $\mathcal{D}$ is denoted $L(\mathcal{D}) = \{\overline{\sigma_I} || \overline{\sigma_O} \in \Sigma^\infty \mid \mathcal{D}(\overline{\sigma_I}) = \overline{\sigma_O}\}$.

Example 2 — Reactive system implementing a simple arbiter.
As an example, consider a simple arbiter that coordinates the access of some shared resource for two clients. The clients send requests *for the shared resource to the arbiter with the input propositions $\{r_a, r_a\}$; whenever proposition $r_c = \top$, then client $c \in \{a, b\}$ requests access. The arbiter provides* grants *to the clients by indicating them with the values of the output propositions $\{g_a, g_b\}$, where $g_c = \top$ indicates that client c can make use of the shared resource. The values of the propositions change during the trace of the arbiter such that the clients can take turn in utilizing the shared resource.*

Figure 2.1 shows an example of a reactive system $\mathcal{D} = (Q, q_0, \Sigma_I, \Sigma_O, \delta, \lambda)$ for the resource arbiter introduced above. The system $\mathcal{D}$ has the following com-

ponents:

- *the input alphabet* $\Sigma_I = \{r_a r_b, \neg r_a r_b, r_a \neg r_b, \neg r_a \neg r_b\}$,
- *the output alphabet* $\Sigma_O = \{g_a g_b, \neg g_a g_b, g_a \neg g_b, \neg g_a \neg g_b\}$,
- *the states* $Q = \{q_{IDLE}, q_{GNT0}, q_{GNT1}\}$ *represent which grant was given last during the execution of the system, where the system being in* q_{IDLE} *denotes that no grant has been given in the previous step of the system's trace,*
- *the transition function* δ *(which, e.g., has* $\delta(q_{IDLE}, \neg r_a \neg r_b) = q_{IDLE}$*), and*
- *the output labeling function* λ *(which, e.g., has* $\lambda(q_{IDLE}, \neg r_a \neg r_b) = \neg g_a \neg g_b$*).*

Serial composition of reactive systems. Let $\mathcal{D} = (Q, q_0, \Sigma_I, \Sigma_O, \delta, \lambda)$ and $\mathcal{D}' = (Q', q_0', \Sigma_I \times \Sigma_O, \Sigma_O, \delta', \lambda')$ 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 $\hat{\mathcal{D}} = \mathcal{D} \circ \mathcal{D}' = (\hat{Q}, \hat{q_0}, \Sigma_I, \Sigma_O, \hat{\delta}, \hat{\lambda})$, 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 $\hat{\lambda}((q, q'), \sigma_I) = \lambda'(q', (\sigma_I, \lambda(q, \sigma_I)))$.

2.4 Automata

The specifications used in this thesis either given as temporal logic specifications (that can be turned into automata), or are given directly as automata.

An *automaton* $\mathcal{A}$ is a tuple $\mathcal{A} = (Q, q_0, \Sigma, \delta, Acc)$, where Q is a finite set of states, $q_0 \subseteq Q$ is the initial state, $\delta : Q \times \Sigma \to Q$ 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 $q_{i+1} = \delta(q_i, \sigma_i)$. An *acceptance condition* is a predicate $Acc : Q^\omega \to \mathbb{B}$, mapping infinite runs $\overline{q}$ to $\top$ or $\bot$. An automaton $\mathcal{A}$ accepts a trace $\overline{\sigma}$ if its run $\overline{q}$ is accepting ($Acc(\overline{q}) = \top$); its language $L(\mathcal{A})$ consists of the set of traces it accepts.

2.5 Acceptance Conditions

We use automata and temporal logic to specify systems, and we synthesize a system that realizes a given specification using games. For both, automata and games, an *acceptance condition* is a predicate $Acc : Q^\omega \to \mathbb{B}$, mapping infinite runs $\overline{q}$ to $\top$ or $\bot$ (accepting and not accepting, or winning and losing, respectively). Commonly used acceptance conditions are the following:

- The *safety* acceptance condition is $Acc(\overline{q}) = \top$ 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 $\exists i \geq 0 \,.\, q_i \in F$, with $F \subseteq Q$ 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 elements 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 $m = 0$, i.e., it is a predicate $\bigwedge_{i=1}^{n} \mathcal{B}(F_i)$, with $F_i \subseteq Q$.
- A *Streett* acceptance condition with k pairs is a predicate $\bigwedge_{i=1}^{k} \mathcal{B}(E_i) \rightarrow \mathcal{B}(F_i)$.

2.6 Logics and Specifications

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

In formal methods, specifications of reactive systems are typically given as formulas in *temporal logic*. In this thesis, we use *linear temporal logic* (LTL).

2.6.1 Safety and Liveness Specifications

Safety Specifications. A specification φ^s is called a *safety specification* [AS85] if finite traces that do not satisfy φ^s cannot be extended to traces that satisfy φ^s, i.e.,

$$\forall \overline{\sigma} \in \Sigma^{*} . (\overline{\sigma} \nvDash \varphi^s \rightarrow (\forall \overline{\sigma}' \in \Sigma^{\infty} . (\overline{\sigma} \cdot \overline{\sigma}') \nvDash \varphi^s)).$$

Intuitively, a safety specification states that "something bad should 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. Safety specifications can be simple *invariance properties* (such as "the level of a water tank should never fall below 1 liter"), but can also be more complex (such as "whenever a valve is opened, it stays open for at least three seconds").

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 reactive system $\mathcal{D}$ satisfies φ^s if the induced run $\overline{q}$ is accepting, i.e., only safe states are visited in $\overline{q}$. The language $L(\varphi^s)$ is the set of all traces satisfying φ.

Liveness Specifications. A property φ^l defines a *liveness* property [AS85], if every finite trace can be extended in an infinite trace that satisfies φ^l, i.e.,

$$\forall \overline{\sigma} \in \Sigma^{*} . \exists \overline{\sigma}' \in \Sigma^{\omega} . (\overline{\sigma} \cdot \overline{\sigma}') \vDash \varphi^l.$$

Informally, a liveness property stipulates that a "good thing" happens during execution eventually.

A formal characterization for safety and liveness properties is given in terms of a Büchi automaton $\varphi = (Q, q_0, \Sigma, \delta, F)$, where $F \subseteq Q$ is a set of accepting states. Intuitively, the "bad thing" for a Büchi automaton is attempting an undefined transition, and the "good thing" is entering an accepting state infinitely often. A trace $\overline{\sigma}$ satisfies φ if the induced run $\overline{q}$ is accepting, i.e., if states in F occur infinitely often in $\overline{q}$.

2.6.2 Propositional Logic

Propositional logic is a language based on *atomic propositions* which can either be true ($\top$) or false ($\bot$). Let B be a set of variables ranging over the Boolean domain $\mathbb{B} = \{\top, \bot\}$ and let AP be the set of *atomic propositions*. The *syntax* of propositional logic is defined as follows:

$$\begin{aligned} a \in \mathsf{AP} &:= \top \mid \bot \mid b, \\ \phi \in \text{propositional-formulas} &:= a \mid \neg\phi \mid \phi \vee \phi, \end{aligned}$$

for each $b \in B$. As usual we denote $\neg(\neg\phi \vee \neg\psi)$ by $\phi \wedge \psi$ and $\neg(\phi \vee \psi)$ by $\phi \rightarrow \psi$. We will call anything that conforms to this grammar a *propositional formula*. A model for a propositional formula is a mapping that assigns either $\top$ or $\bot$ to each variable. The semantics of the connectives $\neg$, $\wedge$, $\vee$, and $\rightarrow$ are defined as usual.

2.6.3 Linear Temporal Logic

Linear Temporal Logic (LTL) [Pnu77] is one of the most popular logic used to specify reactive systems. LTL reasons over linear traces through time. At each point in time, there is only one future timeline that will occur. Traditionally, that timeline is defined as starting in the current time step and progressing infinitely into the future.

LTL extends Boolean logic by the introduction of temporal operators such as X (next time), $\Box$ (globally/always), $\Diamond$ (eventually), and U (until). LTL formulas are constructed as follows [PP06]:

$$\phi \in \text{LTL-formulas} := \phi \in \text{propositional-formulas} \mid \mathsf{X}\phi \mid \phi\mathsf{U}\phi.$$

As usual, we denote $\top\mathsf{U}\phi$ by $\Diamond\phi$ and $\neg\Diamond\neg\phi$ by $\Box\phi$. Given a set of propositions AP, an linear temporal logic (LTL) formula ϕ describes a language $L(\phi)$ in $(2^{\mathsf{AP}})^\omega$. For a word $\overline{\sigma}_{\mathsf{AP}} = \sigma_0\sigma_1\sigma_2 \ldots \in (2^{\mathsf{AP}})^\omega$, we denote with σ_i the set of propositions that are true in location i. We present an inductive definition of when a formula holds in $\overline{\sigma}_{\mathsf{AP}}$ at time i:

- for $a \in \mathsf{AP}$ we have $\sigma_i \vDash a$ iff $a \in \sigma_i$,
- $\sigma_i \vDash \neg\phi$ iff $\sigma_i \nvDash \phi$,
- $\sigma_i \vDash \phi \vee \psi$ iff $\sigma_i \vDash \phi$ or $\sigma_i \vDash \psi$,

- $\sigma_i \vDash \mathsf{X}\phi$ iff $\sigma_{i+1} \vDash \phi$, and
- $\sigma_i \vDash \phi \mathsf{U} \psi$ iff there exists $k \geq i$ such that $\sigma_k \vDash \psi$ and forall $i \leq j < k$ we have $\sigma_j \vDash \phi$.

For a formula ϕ and a position $j \geq 0$ such that $\sigma_j \vDash \phi$, we say that ϕ holds at position j of $\overline{\sigma}_{\mathsf{AP}}$. If $\sigma_0 \vDash \phi$ we say that ϕ holds on $\overline{\sigma}_{\mathsf{AP}}$ and denote it by $\overline{\sigma}_{\mathsf{AP}} \vDash \phi$. The language $L(\phi)$ describes the set of words that satisfy ϕ.

LTL for reactive systems. To use LTL for specifying a set of allowed traces by a reactive system, the joint alphabet $\Sigma = \Sigma_I \times \Sigma_O$ of the system must be decomposable into $\Sigma = 2^{\mathsf{AP}_I} \times \Sigma_I^{rest} \times 2^{\mathsf{AP}_O} \times \Sigma_O^{rest}$ for some system input and output components Σ_I^{rest} and Σ_O^{rest} that we do not want to reason about in the LTL specification. Then, the LTL formula can use $\mathsf{AP} = \mathsf{AP}_I \cup \mathsf{AP}_O$ as the set of atomic propositions. Given a trace $\overline{\sigma}$, we write $\overline{\sigma}_{\mathsf{AP}}$ to denote a copy of the trace where, in each character, the factors Σ_O^{rest} and Σ_I^{rest} have been stripped away so that $\overline{\sigma}_{\mathsf{AP}} \in (2^{\mathsf{AP}})^\omega$.

Example 3 — LTL specification of a simple arbiter.
Let us consider an example for an LTL specification (for the arbiter introduced above in Example 2, Section 2.3) that we build from ground up. By default, LTL formulas are evaluated at the first element of a trace. The LTL formula r_a holds on a trace $\overline{\sigma}_{\mathsf{AP}} = \sigma_0\sigma_1\sigma_2 \ldots \in (2^{\mathsf{AP}})^\omega$ if and only if $r_a \in \sigma_0$. The next-time operator X allows to look one step into the future, so the LTL formula $\mathsf{X}g_a$ holds if $g_a \in \sigma_1$. We can take the disjunction between the formulas r_a and $\mathsf{X}g_a$ to obtain an LTL formula $(r_a \vee \mathsf{X}g_a)$ which holds for a trace if at least one of r_a or $\mathsf{X}g_a$ hold. We can then wrap $(r_a \vee \mathsf{X}g_a)$ into the temporal operator $\square$ to obtain $\square(r_a \vee \mathsf{X}g_a)$. The effect of adding this operator is that in order for $\overline{\sigma}_{\mathsf{AP}}$ to satisfy $\square(r_a \vee \mathsf{X}g_a)$, the subformula $(r_a \vee \mathsf{X}g_a)$ has to hold at every position in the trace. All in all, we can formalize this description by stating that we have that $\sigma \vDash \square(r_a \vee \mathsf{X}g_a)$ holds if and only if for every $i \in \mathbb{N}$, at least one of $r_a \in \sigma_i$ and $g_a \in \sigma_{i+1}$ holds. Note that the reactive system given in Figure 2.1 does not satisfy the specification $\square(r_a \vee \mathsf{X}g_a)$ along all of its traces. The system induces, for instance, a trace of the form $\overline{\sigma} = (\neg r_a \neg r_b, \neg g_a \neg g_b)^\omega$ that results from staying in the q_{IDLE} state forever, along which this specification is not fulfilled. A specification that is however satisfied along all traces of the system is $\square(\square(r_a \wedge \neg r_b) \rightarrow \Diamond\square g_a)$, which can be read as "If from some point onwards, request r_a is always set to $\top$ while request proposition r_b is not, then eventually, a grant is given to process a for eternity."

2.7 Games

In this section, we introduce basic concepts from game theory that are relevant for us to synthesize a system that realizes a given specification via game solving.

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.

Plays. 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 $\top$.

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.

Strategies. A deterministic (memoryless) *strategy* for the environment is a function $\rho_e : G \to \Sigma_I$. A non-deterministic (memoryless) *strategy* for the system is a relation $\rho_s : G \times \Sigma_I \to 2^{\Sigma_O}$ and a deterministic (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 $q_0 \notin W$. 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.

Wrong traces and outputs. 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*. We will also refer to wrong outputs as *unsafe* outputs and to correct outputs as *safe* outputs.

Comparing strategies. First, we compare non-deterministic winning strategies of the system by comparing the *behaviours* that they allow [BJW02]. 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 [Fae09]. 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.

2.8 Reactive Synthesis from Safety Specifications

Consider a safety specification to be given in the form of a deterministic safety word automaton $\varphi^s = (Q, q_0, \Sigma, \delta, F)$, i.e., an automaton in which only safe states in F may be visited. Reactive synthesis enforces φ^s by solving a *safety game* $\mathcal{G} = (Q, q_0, \Sigma_I, \Sigma_O, \delta, F)$ built from φ^s. Given an explicit representation of the safety specification φ^s as a game graph $\mathcal{G}$, the problem of deciding the realizability of a safety specification is known to be solvable in linear time [Maz01].

The standard textbook algorithm for solving safety games [Tho95] proceeds in two steps. First, compute the winning region $W \subseteq F$ and the winning strategy ρ_s from the safety game $\mathcal{G}$. Second, implement the winning strategy ρ_s in a circuit. Any system implementing ρ_s ensures that unsafe states will never be visited. If no such implementation exists, a synthesis algorithm reports unrealizability.

Since *reactive synthesis is a game*, the true power of synthesis is *planning ahead*: synthesized systems will never allow to visit a state from which the environment can force to visit an unsafe state in the future.

Example 4
Suppose that a car is heading towards a cliff. To enforce that the car never crosses the cliff, it has to be slowed down long before it reaches the cliff, and thus far before an abnormal operating condition such as falling can be detected. In particular, the system has to avoid all states from which avoiding to reach the cliff is no longer possible. These states are outside of the winning region of the safety game.

2.9 Markov Decision Processes

When discussing shields for reinforcement learning, we model the environment as Markov Decision Process (MDP) [Put94].

For a set X, let 2^X denote the power set of X. A *probability distribution* over a finite or countably infinite set X is a function $\mu : X \to [0, 1] \subseteq \mathbb{R}$ with $\sum_{x \in X} \mu(x) = \mu(X) = 1$. The set of all distributions on X is denoted by $Distr(X)$. The set $supp(\mu) = \{x \in X \mid \mu(x) > 0\}$ is the *support* of $\mu \in Distr(X)$.

A *Markov decision process* (MDP) $\mathcal{M} = (S, s_I, A, P, R)$ is a tuple with a finite set S of states, a unique initial state $s_I \in S$, a finite set $A = \{a_1 \dots a_n\}$ of actions, a *probabilistic transition function* $P : S \times A \to Distr(S)$, and an

immediate reward function $R : S \times A \to \mathbb{R}$. The reward function may sometimes be omitted.

MDPs operate by means of *nondeterministic choices* of actions at each state, whose successors are then determined *probabilistically* with respect to the associated probability distribution. The set of *enabled* actions at state $s \in S$ is denoted by $A(s) = \{a \in A \mid P(s,a) \neq 0\}$. To avoid deadlock states, we assume that $|A(s)| \geq 1$ for all $s \in S$. A *cost function* $c\colon S \times A \to \mathbb{R}_{\geq 0}$ for an MDP $\mathcal{M}$ adds a cost to each *transition* $(s,a) \in S \times A$ with $a \in A(s)$.

A *path* in an MDP $\mathcal{M}$ is a finite (or infinite) sequence $\pi = s_0 a_0 s_1 a_1 \ldots$ with $P(s_i, a_i, s_{i+1}) > 0$ for all $i \geq 0$. The set of all paths in $\mathcal{M}$ is denoted by $Paths^{\mathcal{M}}$, and all paths starting in state $s \in S$ are denoted by $Paths^{\mathcal{M}}(s)$. The cost of a finite path π is defined as the sum of the costs of all transitions in π, i.e., $c(\pi) = \sum_{i=0}^{n-1} c(s_i, a_i)$ where n is the number of transitions in π.

If $|A(s)| = 1$ for all $s \in S$, all actions can be disregarded and the MDP $\mathcal{M}$ reduces to a *discrete-time Markov chain (MC)*, yielding a transition probability transition function of the form $P\colon S \to Distr(S)$.

A *policy* is a function $\sigma\colon S^* \to Distr(A)$ with $supp(\sigma(s_1 \ldots s_n)) \subseteq A(s_n)$, where S^* denotes a finite sequence of states. For many specifications, it suffices to consider stationary, deterministic policies $\sigma\colon S \to A$ [Put94]. For multiple—possibly conflicting—specifications, more general policies (with randomization and finite memory) are necessary [CMH06].

For an MDP $\mathcal{M}$, *probabilistic model checking* [Kat16, Kwi03] employs value iteration or linear programming to compute the probabilities of *all states and actions of the MDP* to satisfy an LTL property φ. Tool support is readily available in `PRISM` [KNP11], `Storm` [DJKV17] or `Modest` [HH14]. Specifically, we compute $\eta^{\max}_{\varphi,\mathcal{M}}\colon S \to [0,1]$ or $\eta^{\min}_{\varphi,\mathcal{M}}\colon S \to [0,1]$, which give for all states the minimal (or maximal) probability over all possible policies to satisfy φ. For instance, for φ encoding to reach a set of states F, $\eta^{\max}_{\varphi,\mathcal{M}}(s)$ describes the maximal probability to "eventually" reach a state in F.

2.10 Reinforcement Learning

In *reinforcement learning* (RL) [SB98], an agent must learn behavior through trial-and-error via interactions with an unknown environment. The typical framing of a reinforcement learning scenario is illustrated in Figure 2.2. A learning agent takes an action in an unknown environment. The environment executes the action, moves to a next state, and computes a reward, which is fed back into the agent. The goal of a reinforcement learning agent is to collect as much reward as possible.

More formally, we have the following. The environment is modeled by an MDP $\mathcal{M} = (S, s_I, A, P, R)$. The goal of reinforcement learning is to discover a policy $\Pi : S \to A$ that maximizes the expected long-term cumulative reward. The learning agent and the environment interact in discrete time steps.

- At each time step t, the agent receives an observation of the environment's state s_t and a reward r_t.

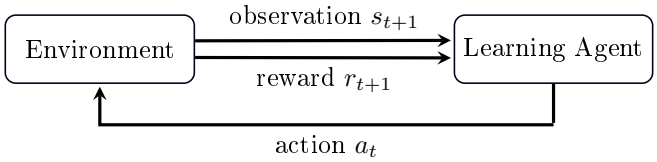


Figure 2.2: Reinforcement learning

- Using s_t and r_t, the agent updates its current policy and chooses a next action $a_t \in A$.
- The environment then moves to a new state s_{t+1} with the probability $P(s_t, a_t, s_{t+1})$ and determines the reward $r_{t+1} = R(s_t, a_t, s_{t+1})$.
- The observation of the state s_{t+1} and the reward r_{t+1} are fed back to the agent. Based on this information, the agent updates its current policy and picks a new action a_{t+1}.

We refer to negative rewards $r_t < 0$ as *punishments*. The *return* $ret = \sum_{t=0}^{\infty} \gamma^t r_t$ is the cumulative future discounted reward, where r_t is the immediate reward at time step t, and $\gamma \in [0, 1]$ is the *discount factor* that controls the influence of future rewards. The objective of the agent is to learn an *optimal policy* $\Pi : S \to A$ that maximizes (over the class of policies considered by the learner) the expectation of the return; i.e. $max_{\pi \in \Pi} E_\pi(ret)$, where $E_\pi(.)$ stands for the expectation with respect to the policy π.

To discover the optimal policy, a reinforcement learning agent has to try actions that it has not selected before. On the one hand, the agent has to exploit what it has already experienced in order to obtain reward, on the other hand it also has to explore in order to make better action selections in the future. This tradeoff between exploration and exploitation is one of the big challenges that arise in reinforcement learning.

In this thesis, we use Q-learning [SB98] as a reinforcement learning algorithm and an ϵ-greedy exploration policy [SB98]. For any finite Markov decision process, Q-learning finds an optimal policy, given infinite exploration time and a partly-random policy. In ϵ-greedy exploration, $0 < \epsilon < 1$ is a parameter controlling the amount of exploration versus exploitation. Exploitation is chosen with probability $1 - \epsilon$, and exploration is chosen with probability ϵ. The parameter ϵ can be a fixed parameter or can be adjusted, making the agent explore progressively less.

Declaration of Sources

Chapter 2 was based on and reuses material from the following sources, previously published by the author:

- [KAB+17] B. Könighofer, M. Alshiekh, R. Bloem, L. R. Humphrey, R. Könighofer, U. Topcu, C. Wang: *Shield synthesis.* FMSD. 2017
- [ABE+18] M. Alshiekh, R. Bloem, R. Ehlers, B. Könighofer, Scott Niekum, and Ufuk Topcu: *Safe reinforcement learning via shielding.* AAAI. 2018
- [JKJB18a] N. Jansen, B. Könighofer, S. Junges, R. Bloem: *Shielded decision-making in MDPs.* ArXiv. 2018

References to these sources are not always made explicit.

3

Post-Posed and Preemptive Shields

3.1 Motivation and Outline

In shield synthesis, we automatically construct a correct-by-construction reactive system that acts as a runtime reinforcement module from a given safety specification. We call such a reactive system a *shield*. A shield can be implemented *before* or *after* the system which has to obey the safety specification.

- If the shield is implemented after the system, we call it a *post-posed* shield.
- If the shield is implemented before the system, we call it a *preemptive* shield.

In both settings, the shield enforces the specified properties at *run time*. The shield continuously monitors the inputs and outputs of the system and interferes with the system only if necessary, and as little as possible, so other non-critical properties are likely to be retained. A shield satisfies the following two properties:

1. *Correctness:* the shielded system satisfies the safety specification.
2. *Minimum interference:* the shield keeps any interferences with the system to a minimum and intervenes only if safe system behavior would be endangered otherwise.

Outline. In this chapter, we will discuss post-posed shielding in Section 3.2 and preemptive shielding in Section 3.3. For both, we will discuss the setting, illustrate the problem with a motivating example, give the definition of the

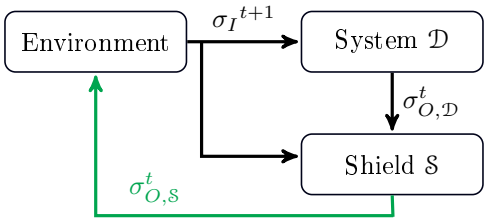


Figure 3.1: Post-posed shielding - Detail

shield, and provide synthesis algorithms to construct such shields. In the following chapters, we will equip these basic shields with additional features to be used for different settings.

3.2 Post-posed Shields

3.2.1 Post-posed Shielding Setting

The post-posed shielding setting is illustrated in detail in Figure 3.1. The shield $\mathcal{S}$ monitors the inputs and outputs of the reactive system $\mathcal{D}$ and substitutes the selected outputs by safe outputs whenever this is necessary to prevent the violation of the safety specification φ. In each step t, the system selects an output $\sigma^t_{O,\mathcal{D}}$. The shield forwards $\sigma^t_{O,\mathcal{D}}$ to the environment, i.e., $\sigma^t_{O,\mathcal{S}} = \sigma^t_{O,\mathcal{D}}$. Only if $\sigma^t_{O,\mathcal{D}}$ is unsafe with respect to φ, the shield selects a different safe output $\sigma^t_{O,\mathcal{S}} \neq \sigma^t_{O,\mathcal{D}}$ instead. The environment executes at $\sigma^t_{O,\mathcal{S}}$, moves to the next state and provides the next input $\sigma_I{}^{t+1}$ to the system and the shield. The system receives $\sigma_I{}^{t+1}$ and picks the next output $\sigma^{t+1}_{O,\mathcal{D}}$ based on that information.

More formally, we have as input alphabet for the shield $\Sigma_{I,\mathcal{S}} = \Sigma_I \times \Sigma_O$, as the shield observes the input from the environment and the output to be corrected from the system. For the output alphabet, we have $\Sigma_{O,\mathcal{S}} = \Sigma_O$, as the shield sends the safe output to the environment. For the input and output alphabet of $\mathcal{D}$, nothing changes when being post-posed shielded, i.e., $\Sigma_{I,\mathcal{D}} = \Sigma_I$ and $\Sigma_{O,\mathcal{D}} = \Sigma_O$.

3.2.2 Illustrative 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. 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 and latency 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.

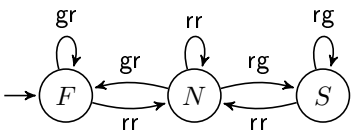


Figure 3.2: Safety specification φ for the traffic light controller.

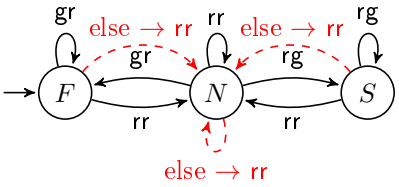


Figure 3.3: A post-posed traffic light shield $\mathcal{S}$.

Suppose 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 defined by the following LTL safety formula:

$$\varphi = \square\neg(\mathsf{gg}) \wedge \square(\mathsf{gr} \rightarrow \mathsf{X}(\mathsf{gr} \vee \mathsf{rr})) \wedge \square(\mathsf{rg} \rightarrow \mathsf{X}(\mathsf{rg} \vee \mathsf{rr})).$$

The specification φ can also be expressed by a safety automaton, which is shown in Figure 3.2. The edges of the automaton are labeled with the controller's outputs for the two roads. There are three non-error states: F indicates 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 is still not a complete specification, the corresponding shield can prevent catastrophic failures.

Figure 3.3 illustrates the behavior of one particular post-posed shield $\mathcal{S}$ for φ (which we could get by using one of the synthesis approaches explained later in this thesis). Intuitively, the states of the shield correspond to the states of the specification automaton. 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 shield forwards the output of the controller to the environment.

Table 3.1 shows how the shields $\mathcal{S}$ corrects a sample output of a traffic light controller. The initial state of the shield is F, i.e., initially, the first road has the green light. In time step 1, the controller sends the output rr, which is accepted by $\mathcal{S}$, and the shield moves to state N. In step 2, the controller sends gg, which violates φ. $\mathcal{S}$ corrects the output to rr and remains in state N. In step 3, the controller gives the output gr, that is accepted by the shield and it moves to

Time Step	1	2	3	4	5	6
Controller	rr	gg	gr	rg	rg	rg
Shield $\mathcal{S}$	rr	rr	gr	rr	rg	rg

Table 3.1: Controller corrected by a post-posed shield $\mathcal{S}$.

state F. In step 4, the controllers gives rg. Since the traffic light cannot toggle from gr to rg according to φ (there is no edge leading from state F to state S), $\mathcal{S}$ changes the output again to rr and moves to state N. Afterwards, the controller sends again rg and $\mathcal{S}$ gives the same output and moves to state S. From here on, $\mathcal{S}$ gives the same output as the controller until the next specification violation.

Let us analyse the behavior of the shield $\mathcal{S}$. First, the shield's output was correct with respect to φ. Second, $\mathcal{S}$ only deviated from the controller in case of a property violation and afterward handed back control to the traffic light controller.

3.2.3 Definition of Post-posed Shields

In this section, we formally define post-posed shields based on its two desired properties: *correctness* and *minimal interference*.

The Correctness Property. By correctness, we refer to the property that the post-posed shield corrects any system's output such that a given safety specification is satisfied.

Definition 1 — Correctness for Post-posed Shields
Let φ be a safety specification, and let $\mathcal{D} = (Q, q_0, \Sigma_I, \Sigma_O, \delta, \lambda)$ and $\mathcal{S} = (Q', q_0', \Sigma, \Sigma_O, \delta', \lambda')$ be two reactive systems. We say that $\mathcal{S}$ ensures correctness *if for any $\mathcal{D}$ it holds that $(\mathcal{D} \circ \mathcal{S}) \vDash \varphi$.*

The Minimal Interference Property. Since a shield must work for any system, the synthesis procedure does not consider the system's implementation. This property is crucial because the system may be unknown or too complex to analyze. On the other hand, the system 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).

The basic requirement for minimal interference is that a post-posed shield can only deviate from the system if a property violation becomes unavoidable. This property can be extended in several ways, e.g., one could require that the shield's output is only allowed to deviate from the system's output as little as possible [BBD+19]. In Chapter 4, we will extend the minimal interference property by requiring that the shield has to change the output in such a way that it can hand back control to the system as soon as possible.

Definition 2 — Minimal Interference for Post-posed Shields
Let φ be a safety specification, let $\mathcal{D} = (Q, q_0, \Sigma_I, \Sigma_O, \delta, \lambda)$ be a reactive system,

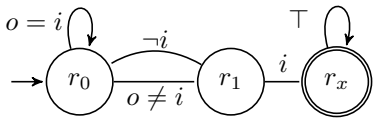


Figure 3.4: The safety automaton φ of Example 5.

and let $\mathcal{S} = (Q', q_0', \Sigma, \Sigma_O, \delta', \lambda')$ *be a reactive system to be attached after* $\mathcal{D}$*, i.e.,* $(\mathcal{D} \circ \mathcal{S})$*. We say that* $\mathcal{S}$ *is* minimal interfering *if for any* $\mathcal{D}$ *and any trace* $\overline{\sigma_I}||\overline{\sigma_O}$ *of* $\mathcal{D}$ *that is correct, 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 3 — Post-posed Shield
Given a specification φ*, a reactive system* $\mathcal{S}$ *is a* post-posed shield *if for any reactive system* $\mathcal{D}$*, it holds that* $(\mathcal{D} \circ \mathcal{S}) \vDash \varphi$ *(Definition 1) and* $\mathcal{S}$ *is* minimal interfering *with* $\mathcal{D}$ *(Definition 2).*

Definition 4 — Post-posed Shielded System
We call the reactive system $\hat{\mathcal{D}} = \mathcal{D} \circ \mathcal{S}$ *a* post-posed shielded system*.*

3.2.4 Synthesis of Post-posed Shields

In this section, we discuss the synthesis procedure to automatically construct post-posed shields from safety specifications via solving a safety game.

Algorithm 1 — Synthesis of Post-posed Shields
Let φ *be the critical safety specification, which is represented as a safety automaton* $\varphi = (Q, q_0, \Sigma, \delta, F)$*. Starting from* φ*, we perform the following two steps to compute a post-posed shield.*

Step 1. Constructing and solving the Safety game $\mathcal{G}$**:** We translate $\varphi = (Q, q_0, \Sigma, \delta, F)$ to a safety game $\mathcal{G} = (G, g_0, \Sigma_I, \Sigma_O, \delta, F)$ with $G = Q$ and $g_0 = q_0$ between two players. In the game, the environment player chooses the inputs $\sigma_I \in \Sigma_I$ and the system chooses the outputs $\sigma_O \in \Sigma_O$. We use standard algorithms for safety games (cf. [Fae09]) to compute the winning region $W \subseteq F$, so that every reactive system $\mathcal{D} \vDash \varphi$ 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 5
Consider the safety automaton φ *in Figure 3.4, 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* $r_0 \to r_1$ *if* $o \neq i$*. In* r_1 *it is too late because visiting an unsafe state cannot be avoided anymore, given that the shield can modify the value of* o *but not* i*.*

Step 2. Translate $\mathcal{G}$ and W to a reactive system $\mathcal{S}$: We translate $\mathcal{G}$ and W to a reactive system $\mathcal{S} = (Q_\mathcal{S}, q_{0,\mathcal{S}}, \Sigma_{I,\mathcal{S}}, \Sigma_{O,\mathcal{S}}, \delta_\mathcal{S}, \lambda_\mathcal{S})$, which constitutes the post-posed shield. The shield has the following components:

- $Q_\mathcal{S} = G$ is the state space,
- $q_{0,\mathcal{S}} = g_0$ is the initial state,
- $\Sigma_{I,\mathcal{S}} = \Sigma_I \times \Sigma_O$ is the input alphabet,
- $\Sigma_{O,\mathcal{S}} = \Sigma_O$ is the output alphabet,
- $\delta_\mathcal{S}$ is the next-state function with

  $$\delta_\mathcal{S}(g, \sigma_I, \sigma_O) = \delta(g, \sigma_I, \lambda_\mathcal{S}(g, \sigma_I, \sigma_O))$$

  for all $g \in G, \sigma_I \in \Sigma_I, \sigma_O \in \Sigma_O$, and
- $\lambda_\mathcal{S}$ is the output function with

$$\lambda_\mathcal{S}(g, \sigma_I, \sigma_O) = \begin{cases} \sigma_O & \text{if } \delta(g, (\sigma_I, \sigma_O)) \in W \\ \sigma_O{}' & \text{if } \delta(g, (\sigma_I, \sigma_O)) \notin W \text{ for some arbitrary} \\ & \text{but fixed } \sigma_O{}' \text{ with } \delta(g, (\sigma_I, \sigma_O{}')) \in W. \end{cases}$$

The shield's output is the output that is actually executed by the environment. Therefore, to determine the next state, the transition function of the shield uses the shield's output $\sigma_{O,\mathcal{S}} = \lambda_\mathcal{S}(q_\mathcal{S}, \sigma_I, \sigma_O)$ and picks next state via $q'_\mathcal{S} = \delta_\mathcal{S}(q_\mathcal{S}, \sigma_I, \sigma_O) = \delta(g, \sigma_I, \sigma_{O,\mathcal{S}})$.

Theorem 1
A reactive system $\mathcal{S}$ constructed according to Algorithm 1 is a post-posed shield (Definition 3).

Proof. We have to proof that $\mathcal{S}$ is *correct* and *minimal interfering*. The output function $\lambda_\mathcal{S}$ ensures correctness (i.e., $\mathcal{D} \circ \mathcal{S} \vDash \varphi$) by always selecting outputs in such a way that only states in the winning region are visited during a play. Hence, $\mathcal{S}$ is correct (Definition 1). At the same time, $\lambda_\mathcal{S}$ keeps the output σ_O of $\mathcal{D}$ intact if σ_O is correct. Hence, $\mathcal{S}$ is minimal interfering (Definition 2). **Q. E. D.**

3.3 Preemptive Shields

3.3.1 Preemptive Shielding Setting

Figure 3.5 depicts the preemptive shielding setting in detail. The interaction between the system, the environment, and the shield is as follows: at every time step t, the shield computes a set of all safe outputs $\sigma^t_{O,\mathcal{S}} = \{\sigma_O{}^{1,t} \dots \sigma_O{}^{k,t}\}$. The

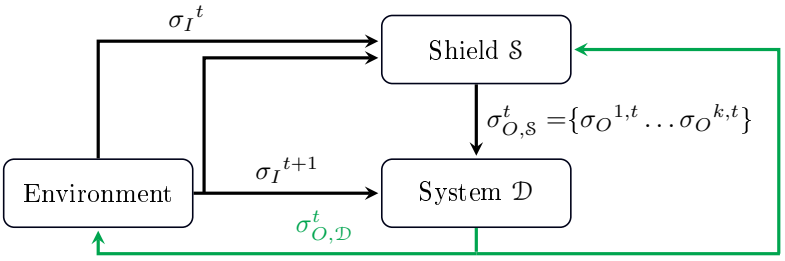


Figure 3.5: Preemptive shielding - Detail

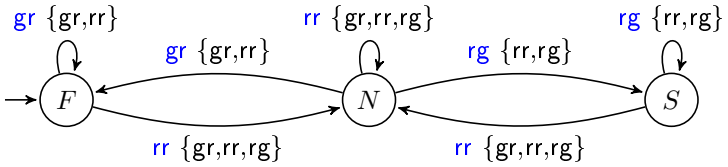


Figure 3.6: A preemptive traffic light shield $\mathcal{S}$.

system receives $\sigma^t_{O,\mathcal{S}}$ from the shield and the input $\sigma_I{}^t$ from the environment, and picks an output $\sigma^t_{O,\mathcal{D}} \in \sigma^t_{O,\mathcal{S}} = \{\sigma_O{}^{1,t} \dots \sigma_O{}^{k,t}\}$ from it. The environment executes $\sigma^t_{O,\mathcal{D}}$, repeats the current input $\sigma_I{}^t$ and provides the next input $\sigma_I{}^{t+1}$. The shield receives $\sigma_I{}^t$, $\sigma^t_{O,\mathcal{D}}$, and $\sigma_I{}^{t+1}$. The shield moves to the next state based on $\sigma_I{}^t$ and $\sigma^t_{O,\mathcal{D}}$ and computes the next set of safe outputs $\sigma^{t+1}_{O,\mathcal{S}} = \{\sigma_O{}^{1,t+1} \dots \sigma_O{}^{k,t+1}\}$ based on its next state and $\sigma_I{}^{t+1}$.

More formally, for a preemptive shield, we have as output alphabet $\Sigma_{O,\mathcal{S}} = 2^{\Sigma_O}$, as the shield outputs the set of safe outputs for the system to choose from for the respective next step. The shield observes the input from the environment $\sigma_I{}^t$, the output chosen by the system σ'^t_O and the next input $\sigma_I{}^{t+1}$ from the environment. So for the input alphabet of the shield, we have $\Sigma_{I,\mathcal{S}} = \Sigma_I \times \Sigma_O \times \Sigma_I$. The input alphabet of the system is $\Sigma_{I,\mathcal{D}} = \Sigma_I \times 2^{\Sigma_O}$ and the output alphabet of the system is $\Sigma_{O,\mathcal{D}} = \Sigma_O$.

3.3.2 Illustrative Example

Let us consider the example of a traffic light controller from Section 3.2.2, and let us apply the preemptive shielding approach on this example.

The behavior of a particular preemptive shield $\mathcal{S}$ for the given traffic light specification φ is illustrated in Figure 3.6. Edges are labeled with the inputs (blue) and outputs (black, set) of the shield. In this example, we do not have any inputs from the environment and the input of $\mathcal{S}$ consists only of the last output of the traffic light controller $\mathcal{D}$ that was already executed by the environment, i.e., $\sigma^t_{I,\mathcal{S}} = \sigma^{t-1}_{O,\mathcal{D}}$. The output of $\mathcal{S}$ constitutes to the set of safe outputs for time step t that is forwarded to $\mathcal{D}$, i.e., $\sigma^t_{I,\mathcal{D}} = \sigma^t_{O,\mathcal{S}}$.

Time Step t	1	2	3	4	5	6
State of $\mathcal{S}$	F	N	F	N	N	S
Input of $\mathcal{S}$ = Output of $\mathcal{D}$ at $t-1$	rr	gr	rr	rr	rg	rr
Next state of $\mathcal{S}$	N	F	N	N	S	N
Output of $\mathcal{S}$	{gr,rr,rg}	{gr,rr}	{gr,rr,rg}	{gr,rr,rg}	{rr,rg}	{gr,rr,rg}
Output of $\mathcal{D}$	gr	rr	rr	rg	rr	rr

Table 3.2: Controller shielded by a preemptive shield $\mathcal{S}$.

Table 3.2 shows how the preemptive shields $\mathcal{S}$ (illustrated in Figure 3.6) ensures correctness in the traffic light setting. The initial state of $\mathcal{S}$ is F. In time step 1, the shield receives $\sigma^1_{I,\mathcal{S}} = \sigma^0_{O,\mathcal{D}} = \mathsf{rr}$ as input, i.e., the last output of the controller $\mathcal{D}$ at $t = 0$ was rr.. Since the last executed action from the environment was rr, the shield moves to state N and allows all outputs except gg, i.e., $\sigma^1_{O,\mathcal{S}} = \{\mathsf{gr}, \mathsf{rr}, \mathsf{rg}\}$. The controller picks the output $\sigma^1_{O,\mathcal{D}} = \mathsf{gr}$ and the environment executes it.

In time step 2, the shield's input is $\sigma^2_{I,\mathcal{S}} = \sigma^1_{O,\mathcal{D}} = \mathsf{gr}$, i.e., gr was the last traffic light phase. The shield moves to state F and outputs $\sigma^2_{O,\mathcal{S}} = \{\mathsf{gr}, \mathsf{rr}\}$ as safe actions. The controller pics $\sigma^2_{O,\mathcal{D}} = \mathsf{gr}$ and forwards it to the environment. In the following steps, the shield continues to monitor the last output of the controller and to provide the list of next safe outputs to the controller.

Let us analyse the behavior of the preemptive shield $\mathcal{S}$. The shielded system was correct, since the controller was only allowed to choose from safe outputs provided by the shield. The shield was minimal interfering with the controller, since it allowed the controller to pick any output as long as it was a safe one.

3.3.3 Definition of Preemptive Shields

As for post-posed shields, we define preemptive shields based on *correctness* and *minimal interference*.

The Correctness Property. At every time step, a preemptive shield computes the set of safe outputs such that the safety specification is satisfied for any output of this set. If the system only chooses outputs that are allowed by the shield, the preemptive shielded system will satisfy the specification.

Definition 5 — Correctness for Preemptive Shields
Let φ be a safety specification, and let $\mathcal{S} = (Q', q'_0, \Sigma_I \times \Sigma_O \times \Sigma_I, 2^{\Sigma_O}, \delta', \lambda')$ and $\mathcal{D} = (Q, q_0, \Sigma_I \times 2^{\Sigma_O}, \Sigma_O, \delta, \lambda)$ be two reactive systems. We say that $\mathcal{S}$ ensures correctness *if for any $\mathcal{D}$ it holds that $(\mathcal{S} \circ \mathcal{D}) \vDash \varphi$.*

The Minimal Interference Property. The minimal interference property in case of preemptive shielding is simpler than for the post-posed shielded setting.

To ensure minimum interference, a preemptive shield has to allow any output as long as it is safe.

Definition 6 — Minimal Interference for Preemptive Shields
Given a safety specification φ, a reactive system $\mathcal{S}$ is minimal interfering *if for any trace $\overline{\sigma_I}||\overline{\sigma_O}$ that is correct, we have that $\overline{\sigma_O} \in \mathcal{S}(\overline{\sigma_I}||\overline{\sigma_O})$.*

Definition 7 — Preemptive Shield
Given a specification φ, a reactive system $\mathcal{S}$ is a preemptive shield *if for any reactive system $\mathcal{D}$ that gives only outputs enabled by $\mathcal{S}$, it holds that $(\mathcal{S} \circ \mathcal{D}) \vDash \varphi$ (Definition 5) and $\mathcal{S}$ is* minimal interfering *with $\mathcal{D}$ (Definition 6).*

Definition 8 — Preemptive Shielded System
We call the reactive system $\hat{\mathcal{D}} = \mathcal{S} \circ \mathcal{D}$ a preemptive shielded system.

3.3.4 Synthesis of Preemptive Shields

We now detail the synthesis procedure to synthesize preemptive shields.

Algorithm 2 — Synthesis of Preemptive Shields
Starting from $\varphi = (Q, q_0, \Sigma, \delta, F)$, we perform the following steps to compute a preemptive shield:

Steps 1. Constructing and solving the Safety game $\mathcal{G}$: Perform as step 1 in Section 3.2.4.

Step 2. Translate $\mathcal{G}$ and W to a reactive system $\mathcal{S}$: We translate $\mathcal{G} = (G, g_0, \Sigma_I, \Sigma_O, \delta, F)$ and W to a reactive system $\mathcal{S} = (Q_\mathcal{S}, q_{0,\mathcal{S}}, \Sigma_{I,\mathcal{S}}, \Sigma_{O,\mathcal{S}}, \delta_\mathcal{S}, \lambda_\mathcal{S})$ that constitutes the preemptive shield. The shield has the following components:

- $Q_\mathcal{S} = G$ is the state space,
- $q_{0,\mathcal{S}} = g_0$ is the initial state,
- $\Sigma_{I,\mathcal{S}} = \Sigma_I \times \Sigma_O \times \Sigma_I$ is the input alphabet,
- $\Sigma_{O,\mathcal{S}} = 2^{\Sigma_O}$ is the output alphabet,
- $\delta_\mathcal{S}$ is the next-state function with

  $$\delta_\mathcal{S}(g, \sigma_I, \sigma_O) = \delta(g, \sigma_I, \sigma_O)$$

  for all $g \in G, \sigma_I \in \Sigma_I, \sigma_O \in \Sigma_O$, and
- $\lambda_\mathcal{S}$ is the output function with

  $$\lambda_\mathcal{S}(g, \sigma_I, \sigma_O, \sigma_I') = \{\sigma_O' \in \Sigma_O \mid \delta(\delta(g, \sigma_I, \sigma_O), \sigma_I', \sigma_O') \in W\}$$

  for all $g \in G$, $\sigma_I \in \Sigma_I$, and $\sigma_O \in \Sigma_O$.

For selecting the next transition, the preemptive shield makes use of the output of the system, i.e., the next transition is chosen via the last observed input from the environment and the last output of the system. The next output of the shield contains all outputs σ'_O, that lead from the current state of the shield $g' = \delta(g, \sigma_I, \sigma_O)$ and the current input of the environment σ'_I to another state in the winning region, i.e., σ'_O is safe.

Theorem 2
A reactive system $\mathcal{S}$ constructed according to Algorithm 2 is a preemptive shield (Definition 18).

Proof. λ_S of $\mathcal{S}$ deactivates all actions that would lead to states outside of the winning region. Therefore, only safe states are visited during the play, and $\mathcal{S}$ is correct (i.e., $\mathcal{S} \circ \mathcal{D} \vDash \varphi$).

Due to the construction of $\mathcal{S}$, an output is only deactivated if taking this output would lead to a state outside of the winning region. From any state that is not winning, the shield cannot ensure that φ will not be violated in the future. Hence, the output needs do be deactivated, and $\mathcal{S}$ is minimal interfering.
Q. E. D.

Declaration of Sources

Chapter 3 was based on and reuses material from the following sources, previously published by the author:

- [BKKW15] R. Bloem, B. Könighofer, R. Könighofer, C. Wang: *Shield synthesis - runtime enforcement for reactive systems.* TACAS. 2015
- [KAB+17] B. Könighofer, M. Alshiekh, R. Bloem, L. R. Humphrey, R. Könighofer, U. Topcu, C. Wang. *Shield synthesis.* FMSD. 2017
- [ABE+17] M. Alshiekh, R. Bloem, R. Ehlers, B. Könighofer, Scott Niekum, and Ufuk Topcu. *Safe reinforcement learning via shielding.* arXiv. 2017
- [ABE+18] M. Alshiekh, R. Bloem, R. Ehlers, B. Könighofer, Scott Niekum, and Ufuk Topcu. *Safe reinforcement learning via shielding.* AAAI. 2018

References to these sources are not always made explicit.

4

k-Stabilizing and Admissible Shields

4.1 Motivation and Outline

In this chapter, we will discuss two post-posed shield synthesis approaches for reactive systems. We call the resulting shields *k-stabilizing* and *admissible* shields. In Chapter 3, we discussed post-posed shields with the minimum interference property requiring that the shield cannot deviate from the system's output before a specification violation by the system is unavoidable. k-stabilizing and admissible shields extend the minimal inference property by providing guarantees on the recovery time:

- k-stabilizing shields guarantee recovery in a finite time, and
- admissible shields attempt to work with the system to recover as soon as possible.

The challenge in recovering quickly lies in the fact that in shield synthesis we treat the system as a *black box*. Therefore, the shield has no information about the current state of the system, nor any information about future inputs and outputs of the system. Nevertheless, the shield should avoid unnecessarily large deviations.

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 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.

Both, k-stabilizing shields and admissible shields enforce critical *safety* properties and ensure minimum interference, 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 interference property of shields, a shield would have to delay enforcing a liveness property as long as possible. 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 system satisfies them. Therefore, we discuss an extension of the k-stabilizing shield synthesis procedure that allows *liveness-preserving* corrections of the system's output.

Outline. The remainder of this chapter is organized as follows. First, we will discuss k-stabilizing shields in Section 4.2 and admissible shields in Section 4.3. For both, we will start with an illustrative example to show the technical challenges and our solution approach. Next, we will define both types of shields and provide synthesis procedures to construct them. In Section 4.4, we will discuss *liveness-preserving* k-stabilizing shields. We will conclude this chapter with experimental results in Section 4.5.

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

Table 4.1: Controller shielded by $\mathcal{S}_A$.

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

Table 4.2: Controller shielded by $\mathcal{S}_B$.

4.2 k-Stabilizing Shields

4.2.1 Illustrative Example

Consider the specification φ of the traffic light example of Section 3.2.2.

Table 4.1 and Table 4.2 show how two different post-posed shields ($\mathcal{S}_A$ and $\mathcal{S}_B$, respectively) correct a sample output of a traffic light controller. Let us first consider Table 4.1. 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`. $\mathcal{S}_A$ 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, $\mathcal{S}_A$ changes the output to `rr`. Afterwards, the controller again sends `gr`. $\mathcal{S}_A$ can end the deviation and to pass on outputs from the controller until the next specification violation.

According to our definition of a post-posed shield, $\mathcal{S}_A$ is a shield: (1) $\mathcal{S}_A$ enforced correctness, and (2) did not deviate unnecessarily. Regarding the recovery time, the shield ended deviation after two steps before handing back control to the traffic light controller.

In Table 4.2, we use the shield $\mathcal{S}_B$. In step 2, $\mathcal{S}_B$ corrects the controller's output 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 $\mathcal{S}_A$ and $\mathcal{S}_B$, $\mathcal{S}_B$ ends the deviation phase faster than $\mathcal{S}_A$. Hence, we prefer the behavior induced by $\mathcal{S}_B$. The behavior of the shield $\mathcal{S}_B$ corresponds with the shield of Figure 3.3.

The challenge in synthesizing shields that recover quickly lies in the fact that we do not know the future inputs/outputs of the system. The question is, without knowing what the future inputs/outputs are, how should the shield correct bad behavior of the system to avoid unnecessarily large deviation in the future? For instance, in step 2, the correction of the shield $\mathcal{S}_A$ was suboptimal since it caused a deviation for two steps instead of one.

4.2.2 Definition of k-stabilizing Shields

We assume that through transmission errors, an arbitrary number of correct outputs by the system $\mathcal{D}$ 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 system's outputs for at most k consecutive

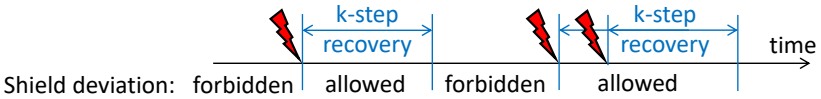


Figure 4.1: Recovery phases of k-stabilizing shields.

time steps, including the current step. This is illustrated in Figure 4.1. Wrong outputs are indicated by lightning.

We will now define k-stabilizing shields.

Definition 9 — k-Stabilization of Traces
Let φ be a safety specification, and let $\overline{\sigma} = (\overline{\sigma_I}||\overline{\sigma_O}) \in \Sigma^\omega$ be a correct trace. A post-posed 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 $(\overline{\sigma_I}[0 \dots i-1]||\overline{\sigma_O}[0 \dots i-1]) \cdot (\overline{\sigma_I}[i], \overline{\sigma_O}[i]^f)$ is wrong and $E = \{i \mid \overline{\sigma_O}[i] \neq \overline{\sigma_O}^f[i]\}$, we have*

$$\overline{\sigma_O}^\star = \mathcal{S}(\overline{\sigma_I}, \overline{\sigma_O}^f),$$

$$(\overline{\sigma_I}||\overline{\sigma_O}^\star) \vDash \varphi \text{ and,}$$

$$\forall j \,.\, \overline{\sigma_O}^\star[j] \neq \overline{\sigma_O}^f[j] \rightarrow \exists i \in E.j - i \leq k.$$

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 system $\overline{\sigma_O}^f$ 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 10 — k-Stabilizing Shields
A post-posed 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 system 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 system 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 system 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 system 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 system, i.e., even if $\overline{\sigma_O}^\star \neq \overline{\sigma_O}$.

4.2.3 Synthesis of k-stabilizing Shields

The flow of our synthesis procedure is illustrated in Figure 4.2.

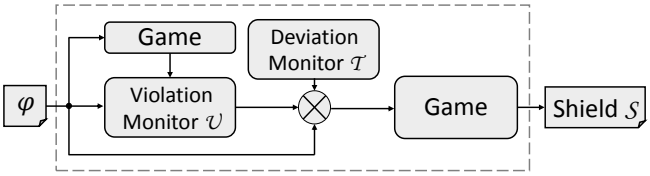


Figure 4.2: Outline of the shield synthesis procedure for k-stabilizing and admissible shields.

Algorithm 3 — Synthesis of k-stabilizing Shields
Let φ be the critical safety specification, which is represented as a safety automaton $\varphi = (Q, q_0, \Sigma, \delta, F)$. Starting from φ, our k-stabilizing shield synthesis procedure consists of three 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 system. 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 system. The violation monitor $\mathcal{U}$ observes the system from all states the system could reach under the current input and a correct output. Note that when multiple states are being monitored if the system's output is wrong from all monitored states, $\mathcal{U}$ monitors all states the system could reach from all currently monitored states under the current input. If the system's output is correct from one or more currently monitored states, it only continues monitoring states reachable from those monitored states under the system'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 $\mathcal{D} \vDash \varphi$ 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.

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 system 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 $\bot$.

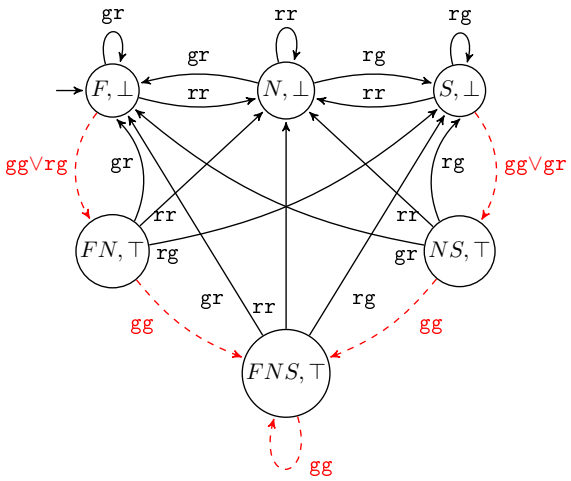


Figure 4.3: Violation monitor $\mathcal{U}$ of Example 6.

From φ, the final violation monitor is $\mathcal{U} = (U, u_0, \Sigma^u, \delta^u)$, such that:

- $U = (2^Q \times \{\top, \bot\})$ is the state space,
- $u_0 = (\{q_0\}, \bot)$ is the initial state,
- $\Sigma^u = \Sigma_I \times \Sigma_O^u$ is the input/output alphabet with $\Sigma_O^u = 2^{O \cup z}$, and
- δ^u is the next-state function 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 system 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 system.

Example 6

We illustrate the construction of $\mathcal{U}$ using the specification φ from Figure 3.2,

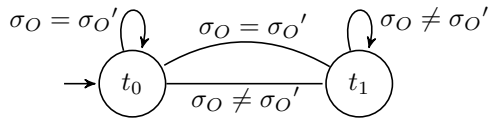


Figure 4.4: The deviation monitor $\mathcal{T}$.

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 Figure 4.3. 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 system 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 Figure 4.3. If the current state is $(\{F\}, \bot)$ and $\mathcal{U}$ observes the output `gg` *from the system, 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 states 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 system from all states that are reachable from the current set of monitored states: in this case, the set of monitored states are 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 system'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, $\mathcal{T}$ will be in t_0. The deviation monitor is shown in Figure 4.4.

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 phase only, and the output of the shield is always correct.

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

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

- F^s is the set of safe states with

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

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. [Fae09]) 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 ($\mathcal{D} \circ \mathcal{S} \vDash \varphi$) and keeps the output of the system $\mathcal{D}$ 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 7) 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,
- $g_0^b = g_0^s$ is the initial state,
- $\Sigma_I^b = \Sigma_I^s$ and $\Sigma_O^b = \Sigma_O^s$ is the input/output alphabet which 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. [Maz01]) 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 $\delta'(g, \sigma_I) = \delta^b(g, \sigma_I, \rho^b(g, \sigma_I))$. Otherwise, the synthesis problem is not realizable.

Theorem 3
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 ($\mathcal{D} \circ \mathcal{S} \vDash \varphi$) and that $\mathcal{S}$ does not deviate from $\mathcal{D}$ unnecessarily. Therefore, $\mathcal{S}$ is a post-posed shield (see Definition 3). 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 10). **Q. E. D.**

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 4
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 $\mathcal{O}(|Q|^3 \cdot 2^{|Q|})$ 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 [Maz01]. **Q. E. D.**

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 system is irrelevant for the shield synthesis procedure.

4.3 Admissible Shields

The proposed k-stabilizing shield synthesis approach (Section 4.2) has a limitation with a significant impact in practice: the k-stabilizing shield synthesis problem is *unrealizable* for many safety-critical systems, because a finite number of deviations cannot be guaranteed. In this section, we will introduce *admissible* shields, which improve k-stabilizing shields in the following way: whereas k-stabilizing shields take an *adversarial* view on the system, admissible shields take a *collaborative* view. That is if there is no shield that guarantees recovery

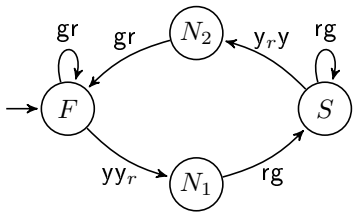


Figure 4.5: Safety specification for the traffic light controller with 4 phases.

within k steps regardless of system behavior (for any finite k), the admissible shield will attempt to work with the system to recover as soon as possible.

4.3.1 Illustrative Example

Let us extend the simple traffic light controller of Section 3.2 to a controller for a traffic light with three colors and four phases. For both roads the phases are red (r), yellow (y), green (g), and red-and-yellow (y$_r$). The safety properties encode the following.

1. The output gg is never allowed.
2. The output cannot change from gr to rg without passing yy$_r$.
3. The output cannot change from rg to gr without passing y$_r$y.
4. The output yy$_r$ is always followed by rg, and the output y$_r$y is always followed by gr.

These properties can be formulated by the following LTL formula φ :

$$\varphi = \square\neg(\mathsf{gg}) \wedge \square(\mathsf{gr} \rightarrow \mathsf{X}(\mathsf{gr} \vee \mathsf{yy}_r)) \wedge \square(\mathsf{rg} \rightarrow \mathsf{X}(\mathsf{rg} \vee \mathsf{y}_r\mathsf{y})) \wedge \square(\mathsf{yy}_r \rightarrow \mathsf{Xrg}) \wedge \square(\mathsf{y}_r\mathsf{y} \rightarrow \mathsf{Xgr}).$$

Figure 4.5 shows the specification φ expressed by a safety automaton. The edges of the automaton are labeled with the controller's outputs for the two roads. There are four non-error states: F denotes the state where the first road has the green light, and S denotes the state where the second road has the green light. In N_1, the first road has the yellow light, and the second road has the red and yellow light. In N_2, the first road has the red and yellow light, and the second road has the yellow light.

Using a k-stabilizing shield. Table 4.3 shows how a post-posed shield $\mathcal{S}$ corrects a sample output of a traffic light controller. Initially, the first road has the green light, and the controller sends the output rg to keep it that way, which is accepted by the shield. In step 2, the controller sends gg, which violates φ.

Time Step	1	2	3	4	5	6	7
Controller	gr	**gg**	rg	$y_r y$	gr	yy_r	rg
Shield $\mathcal{S}$	gr	**gr**	yy_r	rg	yy_r	gr	$y_r y$

Table 4.3: Controller corrected by a post-posed shield $\mathcal{S}$.

To enforce correctness, a shield could either overwrite the output to yy_r or gr. $\mathcal{S}$ corrects the output to gr, and the first road keeps the green light. In step 3, the output of the controller is rg. Since φ does not allow to switch from gr to rg directly, the shield has to alter the output again and chooses yy_r. In step 4, the controller commands to change the phase of the traffic light again, this time to yy_r. The shield deviates again and gives rg, and so on.

Therefore, a single specification violation can lead to an infinitely long deviation between the controller's output and the shield's output. A k-stabilizing shield is allowed to deviate from the controllers outputs for at most k consecutive time steps. Hence, no k-stabilizing shield exists.

Using an admissible shield. Recall the situation in which the shield caused the actual traffic light phase to "fall behind" the traffic light phase requested by the controller. The shield should then implement a best-effort strategy to "synchronize" the actual traffic light phase with the phase commanded by the controller. Though this cannot be guaranteed, the controller is not adversarial towards the shield, so the shield will likely be possible to achieve this re-synchronization, for instance when the controller keeps the same phase for several time steps. This possibility motivates the concept of an admissible shield. For instance, if in step 7 the controller sends the output rg, the shield will be able to catch up and to end the deviation by the next specification violation.

4.3.2 Definition of Admissible Shields

In this section, we discuss admissible shields. We distinguish between two situations. In states of the system in which a finite number k of deviations can be guaranteed, an admissible shield takes an *adversarial* view on the system: 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 system 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 system acts cooperatively.

Definition 11 — Adversely Subgame Optimal Shield
A post-posed shield $\mathcal{S}$ is adversely subgame optimal *if, for any trace $\overline{\sigma} \in \Sigma^*$, $\mathcal{S}$ adversely $k-$stabilizes $\overline{\sigma}$ (Definition 9) 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 12 — Collaborative k-Stabilization of Traces
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 $(\overline{\sigma_I}[0\dots i-1]||\overline{\sigma_O}[0\dots i-1]) \cdot (\overline{\sigma_I}[i], \overline{\sigma_O}[i]^f)$ is wrong and let $E = \{i \mid \overline{\sigma_O}[i] \neq \overline{\sigma_O}^f[i]\}$ be the set of indices of the wrong outputs.

A shield $\mathcal{S}$ collaboratively *k-stabilizes $\overline{\sigma}$ if or 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.,$\overline{\sigma}[0\dots i-1] \cdot (\overline{\sigma_I}[i], {\sigma_O}^c)$ is correct), there exists a correct trace $(\overline{\sigma_I}^c||\overline{\sigma_O}^c) \in \Sigma^\omega$ (i.e., $\overline{\sigma}[0\dots i-1] \cdot (\overline{\sigma_I}[i], {\sigma_O}^c) \cdot (\overline{\sigma_I}^c||\overline{\sigma_O}^c)$ is correct), such that*

$$\overline{\sigma}^{\#} \coloneqq \overline{\sigma}[0\dots i]^f \cdot (\overline{\sigma_I}^c||\overline{\sigma_O}^c),$$
$$\overline{\sigma_O}^{\star} \coloneqq \mathcal{S}(\overline{\sigma}^{\#}),$$
$$(\overline{\sigma_I}||\overline{\sigma_O}^{\star}) \vDash \varphi, \quad \textit{and}$$
$$\forall j \geq i \,.\, \overline{\sigma_O}^{\star}[j] \neq \overline{\sigma_O}^{\#}[j] \rightarrow j - i \leq k.$$

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 Section 4.2.2. 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 13 — 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 14 — 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 15 — 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 system $\mathcal{D}$ 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 system's inputs and outputs such that deviations end after l steps, for the smallest l possible.

4.3.3 Synthesis of 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 Figure 4.2. To synthesize an admissible shield, a Büchi game is constructed in the same way as for k-stabilizing shields. The difference lies in the computation of the strategy of the Büchi game: for k-stabilizing shields, we compute a winning strategy of the Büchi game, and for admissible shields, we compute an admissible strategy.

Algorithm 4 — Synthesis of Admissible Shields
Given is a safety specification $\varphi = (Q, q_0, \Sigma_I \times \Sigma_O, \delta, F)$. Starting from φ, our admissible shield synthesis procedure is as follows:

Steps 1-4. Perform as in Section 4.2.3.

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 system.

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 [Fae09]:

1. Compute the winning region W^b and a winning strategy ρ_w^b for $\mathcal{G}^b$ (cf. [Maz01]).
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

$$(g, (\sigma_I, \sigma_O), g') \in \delta_1^b \begin{cases} \text{if } (g, \sigma_I, \sigma_O) \in \rho_w^b, \text{ or} \\ \text{if } \forall {\sigma_O}' \in \Sigma_O^b . (g, \sigma_I, {\sigma_O}') \notin \rho_w^b \wedge (g, (\sigma_I, \sigma_O), g') \in \delta^b. \end{cases}$$

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 [Fae09]. 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 [Fae09].

Theorem 5
A shield that implements the admissible strategy ρ^b in the Büchi game $\mathcal{G}^b =$

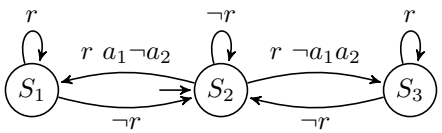


Figure 4.6: Safety specification φ^s of simple arbiter.

$(G^b, g_0^b, \Sigma_I^b, \Sigma_O^b, \delta^b, F^b)$ *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 3. $\mathcal{S}$ is an adversely subgame optimal shield (see Definition 11) for all states of the system in which a finite number of deviations can be guaranteed. This is due to the fact that 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 14), 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 15). **Q. E. D.**

4.4 Liveness-Preserving k-stabilizing Shields

Reactive systems usually not only satisfy safety properties, but are also expected to satisfy liveness properties, 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 system (without shielding).

Definition 16 — Liveness-Preserving Shield
A post-posed shield preserves a given set of liveness properties if any liveness properties satisfied by the system without shielding are also satisfied by the shielded system.

In this section, we discuss an extension to the k-stabilizing shield synthesis procedure that allows liveness-preserving shielding.

4.4.1 Illustrative Example

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., $\square\lozenge(a_1)$.
- φ_2^l: resource 2 has to be granted infinitely often, i.e., $\square\lozenge(a_2)$.

Time Step	1	2	3	4	5
Arbiter	$\neg r \neg a_1 \neg a_2$	**r $a_1 a_2$**	$\neg r \neg a_1 \neg a_2$	**r $a_1 a_2$**	...
Shield $\mathcal{S}$	$\neg r \neg a_1 \neg a_2$	**r $\neg a_1 a_2$**	$\neg r \neg a_1 \neg a_2$	**r $\neg a_1 a_2$**	...

Table 4.4: Shield $\mathcal{S}$ correcting the arbiter.

We attach a post-posed shield to the arbiter to enforce the safety property φ^s expressed by the safety automaton in Figure 4.6. 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 $\neg a_1 a_2$. As usual, the error state is not shown, and 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.

Table 4.4 shows how a post-posed shield $\mathcal{S}$ may correct the arbiter. Initially, the arbiter is in state S_2 and 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 is still in S_2 and receives a request (i.e., r). The arbiter grants both resources at once (i.e., $a_1 a_2$), which violates φ^s. The shield corrects the output to $\neg a_1 a_2$ and monitors the arbiter from state S_1 and S_3. In step 3, the arbiter receives $\neg r$ and sets all outputs to $\bot$. The shield accepts the output, ends the deviation, and monitors the arbiter from state S_2. 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.

4.4.2 Synthesis of Liveness-Preserving k-stabilizing Shields

To construct a system that has all the properties of a k-stabilizing shield and is liveness-preserving, we construct 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 system satisfies all liveness properties, then the shield has to preserve all liveness properties.

Algorithm 5 — Synthesis of liveness-preserving Shields
Let $\varphi^s = (Q^s, q_0^s, \Sigma^s, \delta^s, F^s)$ be the safety specification to be enforced by the shield. Let $\varphi^l = \{\varphi_1^l, \ldots, \varphi_n^l\}$ be the set of liveness properties that if satisfied by the system, have to be preserved after shielding. Each φ_i^l is represented as a Büchi automaton $\varphi_i^l = (Q_i^l, q_{0,i}^l, \Sigma^l, \delta_i^l, F_i^l)$, with F_i^l the set of states that have to be visited infinitely often. The synchronous product φ^l of $\{\varphi_1^l, \ldots, \varphi_n^l\}$ defines an automaton with generalized Büchi acceptance condition $\varphi^l = (Q^l, q_0^l, \Sigma^l, \delta^l, \{F_1^l, \ldots, F_n^l\})$. 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 such a way that the winning strategy corresponds to a k-stabilizing shield.

First (Step 1-a), we construct a Büchi game $\mathcal{G}^b = (G^b, g_0^b, \Sigma_I \times \Sigma_O, \Sigma_O, \delta^b, F^b)$, using the construction described in Section 4.2.3. The resulting 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 system. 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 $\mathcal{G}^{s1} = (G^{s1}, g_0^{s1}, \Sigma_I \times \Sigma_O, \Sigma_O, \delta^{s1}, \langle E_1, F_1 \rangle)$ such that:

- $G^{s1} = G^b$ is the state space,
- $g_0^{s1} = g_0^b$ is the initial state,
- $\delta^{s1} = \delta^b$ is the next-state function, and
- $\langle E_1, F_1 \rangle = \langle G^{s1}, F^b \rangle$ is the GR(1) acceptance condition.

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 $\varphi^l = (Q^l, q_0^l, \Sigma^l, \delta^l, \{F_1^l, \ldots, F_n^l\})$, 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 system satisfies φ^l, then the shield has to satisfy φ^l as well, into the second Streett pair $\langle E_2, F_2 \rangle$, called the liveness-preservation Streett pair. The construction of $\mathcal{G}^{s2}$ consists of two steps.

In the first phase (Step 2-a), we create a GR(1) game $\mathcal{G}^g = (G^g, g_0^g, \Sigma_I \times \Sigma_O, \Sigma_O, \delta^g, Acc)$, such that:

- $G^g = Q^l \times Q^l$ is the state space,
- $g_0 = (q_0^l, q_0^l)$ is the initial state,
- $\Sigma_I \times \Sigma_O$ is the input,
- Σ_O is the output,
- δ^g is the next-state function such that

$$\delta^g\big((q, q'), (\sigma_I, \sigma_O), \sigma_O{}'\big) = \big(\delta^l(q, (\sigma_I, \sigma_O)), \delta^l(q', (\sigma_I, \sigma_O{}'))\big), \text{ and}$$

- Acc is the GR(1) acceptance condition such that

$$Acc = \bigwedge_i \inf(\overline{q}) \wedge F_i^l \neq \emptyset \rightarrow \bigwedge_i \inf(\overline{q'}) \wedge F_i^l \neq \emptyset$$

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 $\mathcal{G}^{s2} = (G^{s2}, g_0^{s2}, \Sigma_I \times \Sigma_O, \Sigma_O, \delta^{s2}, \langle E_2, F_2 \rangle)$ via a counting construction [BCG+10].

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 $\mathcal{G}^{st} = (G^{st}, g_0^{st}, \Sigma_I \times \Sigma_O, \Sigma_O, \delta^{st}, Acc)$ such that:

- $G^{st} = G^{s1} \times G^{s2}$ is the state space,
- $g_0^{st} = (g_0^{s1}, g_0^{s2})$ is the initial state,
- δ^{st} is the next-state function with

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

- Acc is the Streett acceptance condition with

  $$Acc = \{\langle E_1, F_1 \rangle, \langle E_2, F_2 \rangle\}.$$

A winning strategy for the two-pair Streett game $\mathcal{G}^{st}$ corresponds to a liveness-preserving k-stabilizing shield. Streett games with n Streett pairs can be solved using the recursive fixpoint algorithm of [PP06].

4.5 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. This allows the user to specify a set of simple safety properties instead of one complicated property. In the first step, our tool builds the product of all safety automata and construct the violation monitor (see Section 4.2.3 and Section 4.3.3). This step is performed on an explicit representation. For the remaining steps, we use Binary Decision Diagrams (BDDs) for symbolic representation.

We have conducted two sets of experiments, where the benchmarks are (1) selected properties for an ARM AMBA bus arbiter [BJP+12], and (2) selected properties from LTL specification patterns [DAC99]. All experiments were performed on a machine with an Intel i5-3320M CPU@2.6 GHz, 8 GB RAM, and a 64-bit Linux.

4.5.1 A Shield for the ARM AMBA Bus Arbiter

We used properties of an ARM AMBA bus arbiter [BJP+12] 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 [BJP+12], which says that if

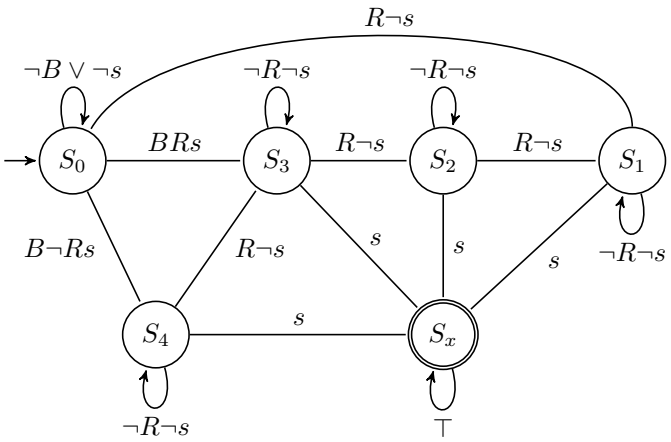


Figure 4.7: Guarantee 3 from the AMBA case study

Step	3	4	5	6	7	8	9	10	11	12
State in Fig. 4.7	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	...

Figure 4.8: Shield execution results

a length-four locked burst access starts, no other access can start until the end of this burst. The safety automaton is shown in Figure 4.7, 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 [BM10] 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 the shield against an arbiter implementation for two bus masters, where we introduced the following 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 Figure 4.7, and going to S_3 instead. An execution trace is shown in Figure 4.8. The first burst starts

Table 4.5: Performance results for AMBA properties

Property	$\|Q\|$	$\|I\|$	$\|O\|$	k	Time[sec]
P1: G1	3	1	1	2	0.1
P2: G1+2	5	3	3	2	0.6
P3: G1+2+3	12	3	3	5	0.26
P4: G1+2+4	8	3	6	2	12
P5: G1+3+4	15	3	5	5	117
P6: G1+2+3+5	18	3	4	5	325
P7: G1+2+4+5	12	3	7	2	66 (admissible)
P8: G2+3+4	17	3	6	5	129 (admissible)
P9: G1+3+4+5	23	3	6	5	786 (admissible)

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, 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 [BJP+12]. Table 4.5 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 synthesis: the table first lists the smallest number of steps under which the shield can 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 three experiments P7-P9, no k-stabilizing shield exists, and the results are given for admissible shields. Both methods run sufficiently fast on all properties.

4.5.2 A Shield for LTL Specification Patterns

Dwyer et al. [DAC99] studied frequently used LTL specification patterns in verification. As an exercise, we applied our tool to the first ten properties from their list [LTL], synthesized k-stabilizing shields, and summarized the results in Table 4.6. 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. Overall, the synthesis method run sufficiently fast on all properties, and the resulting shield size is small. We

Table 4.6: Synthesis results for the LTL patterns.

Nr.	Property	b	$\lvert Q\rvert$	Time[sec]	#Latches	#AIG-Gates
1	$\square\neg p$	-	2	0.01	0	0
2	$\Diamond r \rightarrow (\neg p \mathsf{U} r)$	-	4	0.01	3	10
3	$\square(q \rightarrow \square(\neg p))$	-	3	0.01	2	8
4	$\square((q \wedge \neg r \wedge \Diamond r) \rightarrow (\neg p \mathsf{U} r))$	-	4	0.01	3	15
5	$\square(q \wedge \neg r \rightarrow (\neg p \mathsf{W} r))$	-	3	0.02	3	19
6	$\Diamond p$	0	3	0.01	1	1
6	$\Diamond p$	4	7	0.01	6	30
6	$\Diamond p$	16	19	0.34	10	66
6	$\Diamond p$	64	67	0.67	14	95
6	$\Diamond p$	256	259	39	18	106
7	$\neg r \mathsf{W}(p \wedge \neg r)$	-	3	0.01	5	27
8	$\square(\neg q) \vee \Diamond(q \wedge \Diamond p)$	0	3	0.01	4	19
8	$\square(\neg q) \vee \Diamond(q \wedge \Diamond p)$	4	7	0.02	6	54
8	$\square(\neg q) \vee \Diamond(q \wedge \Diamond p)$	16	19	0.05	10	89
8	$\square(\neg q) \vee \Diamond(q \wedge \Diamond p)$	64	67	0.58	14	114
8	$\square(\neg q) \vee \Diamond(q \wedge \Diamond p)$	256	259	38	18	150
9	$\square(q \wedge \neg r \rightarrow (\neg r \mathsf{W}(p \wedge \neg r)))$	-	3	0.03	5	58

also investigated how the synthesis time increased with an increasingly larger bound b. For the properties P6 and P8, the run time and shield size remained small even for large automata.

Declaration of Sources

Chapter 4 was based on and reuses material from the following sources, previously published by the author:

- [BKKW15] R. Bloem, B. Könighofer, R. Könighofer, C. Wang: *Shield synthesis - runtime enforcement for reactive systems.* TACAS. 2015
- [HKKT16] L. R. Humphrey, B. Könighofer, R. Könighofer, U. Topcu: *Synthesis of admissible shields.* HVC. 2016
- [KAB+17] B. Könighofer, M. Alshiekh, R. Bloem, L. R. Humphrey, R. Könighofer, U. Topcu, C. Wang: *Shield synthesis.* FMSD. 2017

References to these sources are not always made explicit.

5 Explanatory Shields

5.1 Motivation and Outline

In this chapter, 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. We apply a preemptive shield, which restricts the possible actions of the operator. Shielding a human operator requires innovation in the shielding procedure: 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 such shields *explanatory shields*. Explanatory shields are particularly useful in cases in which the specification is very complex, and it isn't very easy for the operator to comprehend why the shield had to interfere.

Outline. In Section 5.2, we will discuss the setting for shielding human-autonomy interactions. We will define explanatory shields in Section 5.3, and we will discuss the synthesis of explanatory shields in Section 5.4. We will conclude this chapter with a case study on UAV mission planning in Section 5.5.

5.2 Explanatory Shielding Setting

Consider a setting in which a human operator controls an autonomous reactive system as part of the environment: in every time step, the environment provides an input (sensor measurements, state information) to the operator. Then the operator selects the next action, and the environment executes the selected action and moves to the next state.

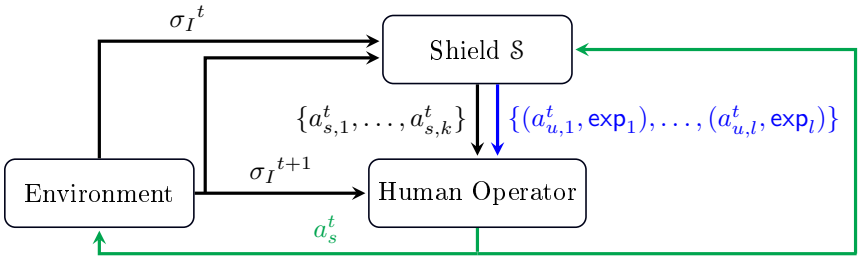


Figure 5.1: Explanatory Shielding Setting

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 $\Sigma_O = A$ is the set of available actions, is met. We modify the loop between the human operator and the environment, as depicted in Figure 5.1. The shield is implemented before the human operator and acts each time the operator is to make a decision. The shield provides a list of safe actions and for each unsafe action, an explanation of why the action is unsafe.

The interaction between the environment, the human operator, and the shield is as follows: at every time step, the shield $\mathcal{S}$ computes a set of all safe actions $\{a^t_{s,1}, \ldots, a^t_{s,k}\}$ and the list of unsafe actions $\{a^t_{u,1}, \ldots, a^t_{u,l}\} = A \setminus \{a^t_{s,1}, \ldots, a^t_{s,k}\}$. Additionally, $\mathcal{S}$ computes for each $a_u \in \{a^t_{u,1}, \ldots, a^t_{u,l}\}$ an explanation exp that explains why a_u is unsafe (we discuss in the next section, how such an explanation looks like). The human operator receives the input from the shield and from the environment. The operator may consider the list of unsafe actions and explanations to understand the restrictions made by the shield. Afterwards, the operator picks an action $a_s \in \{a^t_{s,1}, \ldots, a^t_{s,k}\}$ from the list of safe actions. The environment executes a_s, moves to the next state, and provides the next input to the shield and the operator.

5.3 Definition of Explanatory Shields

Explanatory shields provide a simple diagnosis to the operator and explain why certain actions are unsafe in the current situation. This is particularly helpful if 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 forbid an action. To explain unsafe actions, we propose to use techniques for debugging formal specifications [KHB13].

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 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 17 — 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. Let $\phi \subseteq \varphi$ be a subset of φ with $\phi = (Q', q_0', \Sigma_I \times \Sigma_O, \delta', F')$.

ϕ defines an unsafe core *for a state $q \in Q$, an input $\sigma_I \in \Sigma_I$, and an output $\sigma_O \in \Sigma_O$ if executing $\sigma_I \sigma_O$ from $q' \in Q'$ (q' deduced from $q \in Q$) results in a next state 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 to the problem at hand.

Definition 18 — Explanatory Shield
Given a specification φ, a reactive system $\mathcal{S}$ is an explanatory shield *if $\mathcal{S}$ is a preemptive shield (Definition 18), and it provides the unsafe core for any unsafe action in any situation, (i.e., state-input combination of φ).*

5.4 Synthesis of Explanatory Shields

In this section, we discuss a synthesis approach to construct explanatory shields.

Algorithm 6 — Synthesis of Explanatory Shields
Let $\varphi = \{\varphi_1, \dots, \varphi_l\}$ be the safety specification, where each φ_i is represented as an safety 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 these automata, our explanatory shield synthesis procedure consists of two steps.

Step 1. Compute a Preemptive Shield. Using φ, compute a preemptive shield $\mathcal{S} = (Q_\mathcal{S}, q_{0,\mathcal{S}}, \Sigma_{I,\mathcal{S}}, \Sigma_{O,\mathcal{S}}, \delta_\mathcal{S}, \lambda_\mathcal{S})$ with $\Sigma_{I,\mathcal{S}} = \Sigma_I \times \Sigma_O \times \Sigma_I$ and $\Sigma_{O,\mathcal{S}} = 2^{\Sigma_O}$ via Algorithm 2, presented in Section 3.3.4.

Step 2: Compute the Unsafe Cores. For each state $q \in W$, input σ_I, and unsafe output σ_O, compute the unsafe core $\mathsf{exp}(q, \sigma_I, \sigma_O) \subseteq \varphi$. Unsafe cores are very similar to unrealizable cores [CRST08], 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 [LS08].

To obtain a new reactive system $\mathcal{S}_E = (Q_E, q_{0,E}, \Sigma_{I,E}, \Sigma_{O,E}, \delta_E, \lambda_E)$ that constitutes the explanatory shield, we extend the output function of the preemptive shield $\mathcal{S}$. The explanatory shield has the following components:

- $Q_E = Q_\mathcal{S}$ is the state space,
- $q_{0,E} = q_{0,\mathcal{S}}$ is the initial state,
- $\Sigma_{I,E} = \Sigma_{I,\mathcal{S}}$ is the input alphabet,

- $\Sigma_{O,E} = \Sigma_{O,\mathcal{S}} \times 2^{2^L}$ is the output alphabet with $L = \{1, \ldots, l\}$,
- $\delta_E = \delta_{\mathcal{S}}$ is the next-state function, and
- λ_E is the output function with

$$\begin{aligned}\lambda_E(g, \sigma_I, \sigma_O, \sigma_I') = &\lambda_{\mathcal{S}}(g, \sigma_I, \sigma_O, \sigma_I') \cup \\ &\{\sigma_O' \in \Sigma_O \mid \delta(g', \sigma_I', \sigma_O') \notin W\} \cup \\ &\{i \mid \varphi_i \in \mathsf{exp}(q', \sigma_I', \sigma_O')\}\end{aligned}$$

with $g' = \delta(g, \sigma_I, \sigma_O)$ and for all $g \in G$, $\sigma_I \in \Sigma_I$, and $\sigma_O \in \Sigma_O$.

Counterstrategies and Countertraces. Additionally to providing the unsafe core to the operator for each unsafe action in a given situation, an explanatory shield may also provide counterstrategies and countertraces to the operator.

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. This 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. Therefore, we propose to use a heuristic to compute countertraces as presented in [KHB13].

5.5 Experimental Results

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 an explanatory shield can be used to enforce the desired properties and to provide feedback to the operator for any restrictions on the commands available.

To begin, note that a common UAV control architecture consists of a ground control station that communicates with an autopilot onboard the UAV [CCC10]. 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

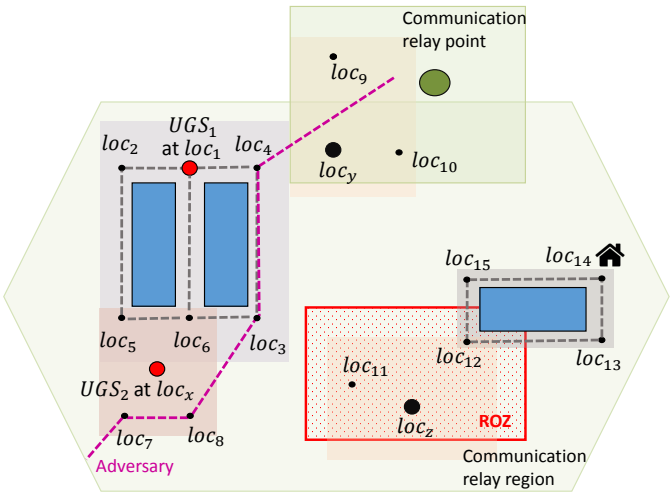


Figure 5.2: A map for UAV mission planning.

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 [DNC10]. 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 Figure 5.2 [FWHT16], 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.

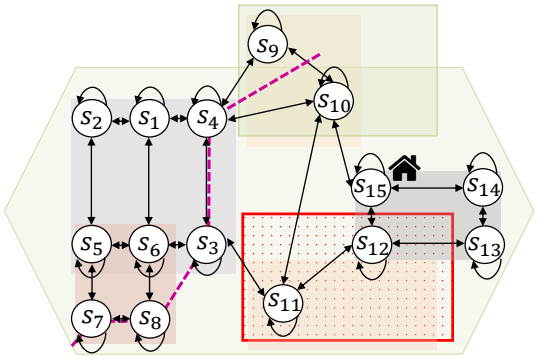


Figure 5.3: Safety automaton of Property P1.

P2. **No communication.** The UAV is not allowed to stay in a zone with reduced communication reliability and has to leave this zone within one time step.

P3. **Restricted operating zones.** The UAV has to leave a ROZ within two time steps.

P4. **Detected by an adversary.** Locations on the adversary's path cannot be visited more than once over any window of three 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 seven steps.

P6. **Refuel.** Once the UAV's battery is low, it should return to a designated landing site at loc_{14} within ten 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 5.3 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 Figure 5.3. The automaton also includes an error state, which is not shown. Missing edges lead to this error state, denoting forbidden situations.

Property	$\|Q\|$	$\|I\|$	$\|O\|$	Time [sec]
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

Table 5.1: Results for UAV experiments

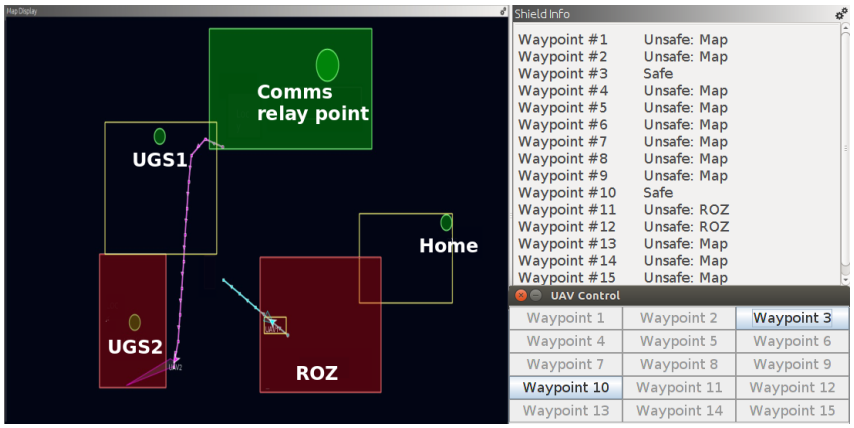


Figure 5.4: Simulation of explanatory preemptive shield developed on AMASE.

For our experiments, we used the six Properties P1-P6 as safety specification $\varphi_1, \ldots, \varphi_6$. We synthesized explanatory shields using the properties $\{\varphi_1, \ldots, \varphi_6\}$ to compute the unsafe cores. All results are summarized in Table 5.1. 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 shield.

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 [Duq09]. AMASE is a flight simulation environment, which models UAVs using a kinematic flight dynamics model that includes environmental effects (e.g., wind) on performance. Figure 5.4 visualizes the map of Figure 5.2 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 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.

Declaration of Sources

Chapter 5 was based on and reuses material from the following sources, previously published by the author:

- [KAB+17] B. Könighofer, M. Alshiekh, R. Bloem, L. R. Humphrey, R. Könighofer, U. Topcu, C. Wang: *Shield synthesis.* FMSD. 2017
- [HKKT16] L. R. Humphrey, B. Könighofer, R. Könighofer, U. Topcu: *Synthesis of admissible shields.* HVC. 2016

References to these sources are not always made explicit.

6

Safe Reinforcement Learning via Deterministic Shields

6.1 Motivation and Outline

In recent years, artificial intelligence (AI) evolved from areas like game-playing or language-translation to critical domains such as health, energy, defense, or transportation. In particular advances in reinforcement learning (RL) enabled a new paradigm for developing controllers for autonomous systems that accomplish complicated tasks in uncertain and dynamic environments. In such an RL-framework, an agent acts to optimize a long-term return that models the desired behavior for the agent and is revealed to it incrementally in a reward signal as it interacts with its environment [SB98].

Increasing use of learning-based controllers in physical systems in the proximity of humans strengthens the concern of whether these systems will operate safely. Particularly during the exploration phase, when an agent chooses random actions to examine its surroundings, it is important to avoid actions that may cause unsafe outcomes.

While convergence, optimality, and data-efficiency of learning algorithms are relatively well understood, safety or more generally correctness of controllers has attracted significantly less attention. For systems employing RL, safety of decision-making, particularly during the exploration phase, is a major open challenge [SSP+17, FZ16, Nat16, RDT16]. The area of *safe exploration* aims to guide RL agents to adhere to safety requirements during this phase [PS14, AOS+16].

We approach the problem of ensuring safety in reinforcement learning from

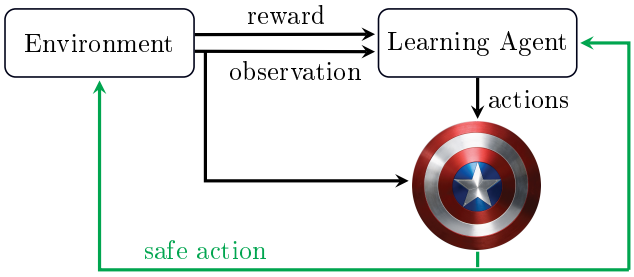


Figure 6.1: Post-posed shielded reinforcement learning

a formal methods perspective, and use our shields to guide the learning agent during exploration. We investigate the question "how can we let a learning agent do whatever it is doing, and also monitor and interfere with its operation whenever absolutely needed to ensure safety?"

In this chapter, we introduce *shielded learning*, a framework that allows applying machine learning to control systems in a way that the *correctness* of the system's execution against a given specification is assured during the learning and controller execution phases.

In the traditional reinforcement learning setting, in every time step, the learning agent chooses an action and sends it to the environment. The environment evolves according to the action and sends the agent an observation of its state and a reward associated with the underlying transition. The objective of the learning agent is to optimize the reward accumulated over this evolution.

Our approach introduces a *shield* into the traditional reinforcement learning setting. The shield is computed upfront from the safety part of the given system specification and an abstraction of the agent's environment dynamics. It ensures safety and minimum interference. As before, with minimum interference, we mean that the shield restricts the agent as little as possible and forbids actions only if they could endanger safe system behavior.

We distinguish between *post-posed shielded learning* and *preemptive shielded learning*, and we modify the loop between the learning agent and its environment in two alternative ways, depending on the location at which the shield is implemented. In the post-posed implementation of the shield, depicted in Figure 6.1, the shield monitors the actions selected by the learning agent and corrects them if and only if the chosen action is unsafe. In the preemptive case, depicted in Figure 6.2, the shield is implemented before the learning agent and acts each time the learning agent is to make a decision and provides a list of safe actions. This list restricts the choices for the learner. The shield provides minimum interference since it allows the agent to follow any policy as long as it is safe.

The goal of shielded learning is to combine the best of two worlds, namely (1) the formal *correctness guarantees* of a controller against a temporal logic specification, as provided by formal methods (and reactive synthesis in particular),

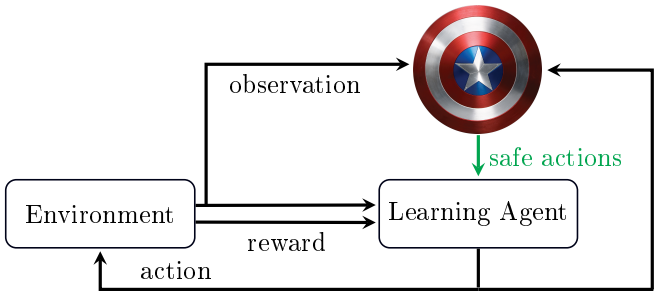


Figure 6.2: Preemptive shielded reinforcement learning

and (2) the *optimality* with respect to an a priori unknown performance criterion, as provided by RL. Additionally, shielded learning offers further pragmatic advantages:

1. *Scalability.* Even though the inner working of learning algorithms is often complex, shielding with respect to critical safety specifications may be manageable. The algorithms we present for the computation of shields make relatively mild assumptions on the input-output structure of the learning algorithm (rather than its inner working). Consequently, the correctness guarantees are agnostic to the learning algorithm of choice.

2. *Separation of concerns.* Our setup introduces a clear boundary between the learning agent and the shield. This boundary helps to separate the concerns, e.g., safety and correctness on one side and convergence and optimality on the other and provides a basis for the convergence analysis of a shielded RL algorithm.

By combining RL with reactive synthesis, we achieve safe reinforcement learning which we define in the following way:

Definition 19 — Safe Reinforcement Learning
Safe RL is the process of learning an optimal policy while satisfying a temporal logic safety specification φ^s during the learning and execution phases.

Outline. This chapter is dedicated to shielded reinforcement learning. Given that in reinforcement learning the environment is often represented as an MDP, we will first discuss how to obtain a deterministic safety word automaton that is a conservative abstraction with respect to the behavior of the real MDP (see Section 6.2). Using this abstraction, we will discuss post-posed shielded learning in Section 6.3 and preemptive shielded learning in Section 6.4. For both, we will discuss the setting and provide synthesis algorithms to construct such shields. Section 6.6 concludes this chapter with experimental results.

6.2 Abstractions

We consider a safety specification to be given in the form of a deterministic safety word automaton $\varphi^s = (Q, q_0, \Sigma, \delta, F)$, i.e., an automaton in which only safe states in F may be visited.

Reactive synthesis does not require the environment dynamics to be completely known in advance. However, to reason about when exactly a specification violation cannot be avoided, we have to give a (coarse finite-state) abstraction of the environment dynamics.

Given that in reinforcement learning the environment is often represented as an MDP, such an abstraction has to be conservative with respect to the behavior of the real MDP. This approximation may have finitely many states, even if the MDP has infinitely many states and/or is only approximately known.

Definition 20 — MDP Abstraction
Given an MDP $\mathcal{M} = (S, s_I, A, P, R)$ and an MDP observer function $f : S \to L$ for some set L, we call a deterministic safety word automaton $\varphi^{\mathcal{M}} = (Q, q_0, \Sigma, \delta, F)$ an abstraction *of $\mathcal{M}$ if*

(i) $\Sigma = A \times L$, *and*

(ii) for every trace $s_0 s_1 s_2 \ldots \in S^\omega$ with the corresponding action sequence $a_0 a_1 \ldots \in A^\omega$ of the MDP, for every automaton run $\overline{q} = q_0 q_1 \ldots \in Q^\omega$ of $\varphi^{\mathcal{M}}$ with $q_{i+1} = \delta(q_i, (l_i, a_i))$ with $l_i = L(s_i)$ for all $i \in \mathbb{N}$, we have that

$$\overline{q} \text{ always stays in } F.$$

An abstraction of an MDP describes how its executions can possibly evolve, and provides the needed information about the environment to allow planning ahead with respect to the safety properties of interest. Without loss of generality, we assume that $\varphi^{\mathcal{M}}$ has no states in F from which all infinite paths eventually leave F. This ensures that paths that model traces that cannot occur in $\mathcal{M}$ are rejected by $\varphi^{\mathcal{M}}$ as early as possible.

For both shielded learning settings we make the following assumptions:

1. The environment can be modeled as an MDP $\mathcal{M} = (S, s_I, A, P, R)$.
2. We have constructed an abstraction $\varphi^{\mathcal{M}} = (Q, q_0, \Sigma, \delta, F)$.
3. The learner accepts elements from $S \times Q$ as state input.

Example 7
We want to learn an energy-efficient controller for a hot water storage tank, depicted in Figure 6.3. Stored water is kept warm by a heater whose energy consumption depends on the filling level of the tank, but we do not know what the exact relationship is. The outflow is always between 0 and 1 liters per second, and the inflow is known to be between 1 and 2 liters per second whenever the valve is open (and it is 0 otherwise). The capacity of the tank is limited to 100 liters, and whenever the inflow is switched on or off, the setting has to be kept

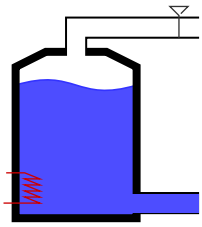


Figure 6.3: A hot water storage tank with an inflow, an outflow, and a tank heater.

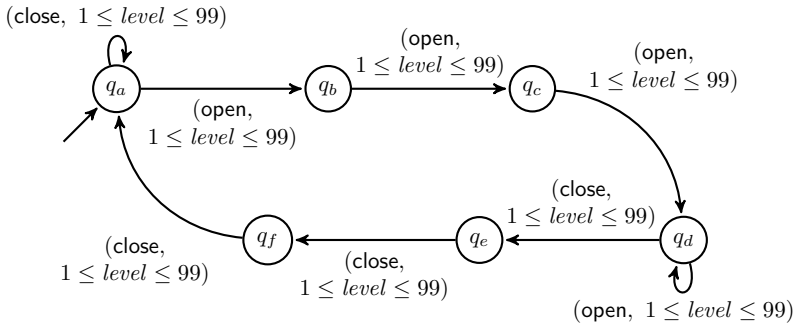


Figure 6.4: The specification φ^s for the water tank controller.

for at least three seconds to limit the wear-out of the valve. Also, the tank must never overflow or run dry.

We can express the safety specification for the water tank valve controller using the following LTL formula:

$$\begin{aligned}\varphi^s = {} & \square(level > 0) \wedge \square(level < 100) \wedge \\ & \square((open \wedge \mathsf{X} close) \rightarrow \mathsf{XX} close \wedge \mathsf{XXX} close) \wedge \\ & \square((close \wedge \mathsf{X} open) \rightarrow \mathsf{XX} open \wedge \mathsf{XXX} open).\end{aligned}$$

The specification consists of four conjuncts, where the first two conjuncts enforce the water levels to be between the minimum and maximum thresholds. The next conjunct enforces that if the valve is open and then closed, then it has to stay closed for two more time steps (seconds). The final conjunct enforces that if the valve is closed and then opened, it has to stay open for two more steps.

We can translate the specification to the safety automaton shown in Figure 6.4. All states are accepting and transitions leading to the error state (which exists in addition to the states in Figure 6.4 and is not accepting) are not shown. It uses the action sets $A = \{\mathsf{open}, \mathsf{closed}\}$ for the inflow valve state, and the label set $L = \{level < 1, 1 \leq level \leq 99, level > 99\}$ as needed information about the water tank filling status. What we know about the behavior of the water tank can be summarized as the abstraction automaton that we give in Figure 6.5.

We will show in Section 6.3 how to compute a post-posed shield from an abstraction automaton and a safety specification automaton. We will then revisit

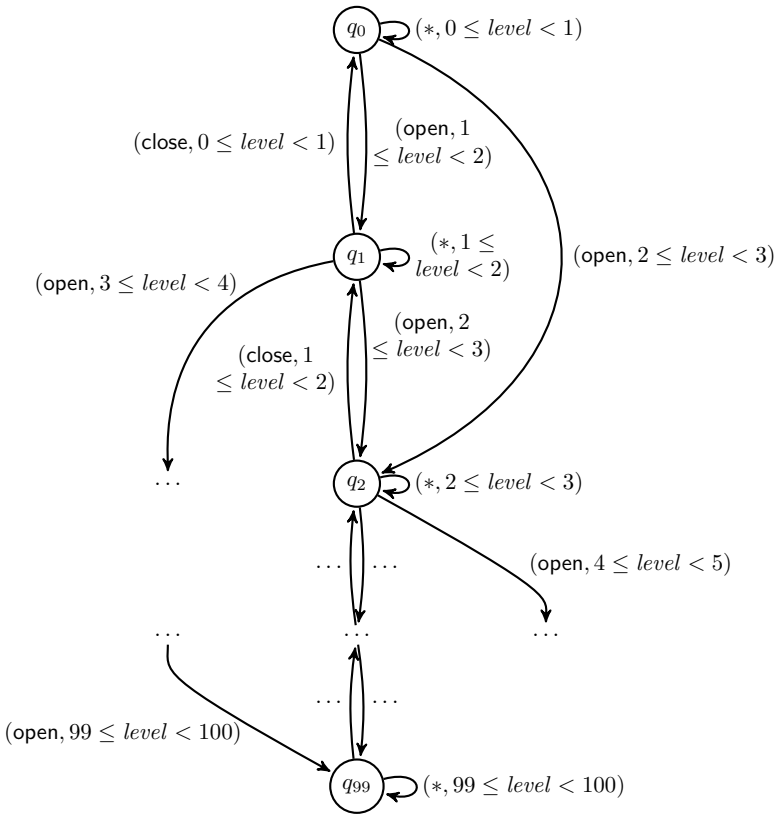


Figure 6.5: The abstraction $\varphi^{\mathcal{M}}$ of the water tank behavior.

this example and give the resulting shield that enforces the specification. The shield will enforce that when the water level in the tank becomes too low, the inflow valve is opened until some minimum level of 4 *is reached, and it will also prevent the inflow from being opened when the level is above* 93*. The latter is necessary as the valve has to stay open for at least three time steps. So as the inflow maybe up to* 2 *liters/second during this time and the outflow may be* 0*, there is otherwise an overflow risk. As the shield is generated using the specification, it plans ahead for this not to happen, so it must prevent the opening of the inflow valve if the level is above* 93*.*

6.3 Post-posed Shields for RL

6.3.1 Post-posed Shielding Setting for RL

The post-posed shielded learning setting is shown in Figure 6.6. The shield monitors the actions of the agent, and substitutes the selected actions by safe

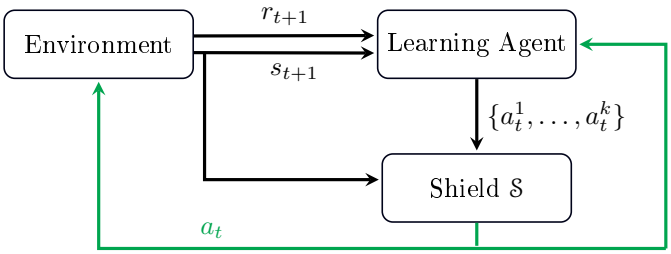


Figure 6.6: Post-posed shielded reinforcement learning - Detail

actions whenever this is necessary to prevent the violation of φ^s.

In detail, we have the following. In each step t, the agent selects an action a_t^1. The shield forwards a_t^1 to the environment, i.e., $a_t = a_t^1$. Only if a_t^1 is unsafe with respect to φ_s, the shield selects a different safe action $a_t \neq a_t^1$ instead. The environment executes a_t, moves to s_{t+1} and provides r_{t+1}. The agent receives a_t and r_{t+1}, and performs policy updates based on that information. For the executed action a_t, the agent updates its policy using r_{t+1}. The question is what the reward for a_t^1 should be in case we have $a_t \neq a_t^1$. We discuss two different approaches.

1. **Assign a punishment r'_{t+1} to a_t^1.** The agent assigns a punishment $r'_{t+1} < 0$ to a_t^1 and learns that selecting a_t^1 at state s_t is unsafe, without ever violating φ^s. However, there is no guarantee that unsafe actions are not part of the final policy. Therefore, the shield has to remain active even after the learning phase.

2. **Assign the reward r_{t+1} to a_t^1.** The agent updates a_t^1 with the reward r_{t+1}. Therefore, picking unsafe actions can likely be part of an optimal policy by the agent. Since an unsafe action is always mapped to a safe one, this does not pose a problem, and the agent never has to learn to avoid unsafe actions. Consequently, the shield is (again) needed during the learning and execution phases.

Properties of Post-Posed Shielding. To be less restrictive to the learning algorithm, we propose that in every time step, the agent provides a ranking $rank_t = (a_t^1, \ldots, a_t^k)$ on the allowed actions, i.e., the agent wants a_t^1 to be executed the most, a_t^2 to be executed the second-most, etc. The ranking does not have to contain all available actions, i.e., $1 \leq |rank_t| \leq n_t$, where n_t is the number of available actions in step t. The shield selects the first action $a_t \in rank_t$ that is safe according to φ^s. Only if all actions in $rank_t$ are unsafe, the shield selects a safe action $a_t \notin rank_t$. Both approaches for updating the policy discussed before can naturally be extended for a ranking of several actions.

A second advantage of having a ranking on actions is that the agent can *perform several policy updates at once*; e.g., if all actions in $rank_t$ are unsafe,

the agent can perform $|rank_t|+1$ policy updates in one step by using the rewards r'_{t+1} or r_{t+1} for all of them, depending on which of the above variants is used.

The big advantage of post-posed shielding is that it works even if the learning algorithm is *already in the execution phase* and therefore follows a fixed policy. In every step, the learning algorithm only sees the state of the MDP (without the state of the shield), and then the shield corrects the learner's actions whenever this is necessary to ensure safe operation of the system. The learning agent does not even need to know that it is shielded.

6.3.2 Synthesis of Post-posed Shields for RL

In this section, we give an algorithm to compute post-posed shields from φ^s and an MDP abstraction $\varphi^{\mathcal{M}}$ that represents the environment in which the agent shall operate. We prove that the computed shields (1) enforce the correctness criterion, and (2) are the minimally interfering shields among those that enforce φ^s on all MDPs for which $\varphi^{\mathcal{M}}$ is an abstraction.

Given is an RL problem in which an agent has to learn an optimal policy for an unknown environment that can be modelled by an MDP $\mathcal{M} = (S, s_I, A, P, R)$ while satisfying a safety specification $\varphi^s = (Q, q_0, \Sigma, \delta, F)$ with $\Sigma = \Sigma_I \times \Sigma_O$ and $A = \Sigma_O$. We assume some abstraction $\varphi^{\mathcal{M}} = (Q_{\mathcal{M}}, q_{0,\mathcal{M}}, A \times L, \delta_{\mathcal{M}}, F_{\mathcal{M}})$ of $\mathcal{M}$ for some MDP observer function $f : S \to L$ to be given. Since φ^s models a restriction of the traces of the MDP and the learner together that we want to enforce, we assume it to have $\Sigma = L \times A$, i.e., it reads the part of the system behavior that the abstraction is concerned with.

Algorithm 7 — Synthesis of Post-posed Shields for RL
Staring from $\varphi^s = (Q, L \times A, \Sigma, \delta, F)$ and $\varphi^{\mathcal{M}} = (Q_{\mathcal{M}}, q_{0,\mathcal{M}}, A \times L, \delta_{\mathcal{M}}, F_{\mathcal{M}})$, we perform the following steps to compute a post-posed shield.

Step 1. Constructing a safety game $\mathcal{G}$: We translate φ^s and $\varphi^{\mathcal{M}}$ to a safety game $\mathcal{G} = (G, g_0, \Sigma_I, \Sigma_O, \delta, F^g)$ between two players. In the game, the environment player chooses the next observations from the MDP state (i.e., elements from L), and the system chooses the next action (i.e., elements from A). Formally, $\mathcal{G}$ has the following components:

- $G = Q \times Q_{\mathcal{M}}$ is the state space,
- $g_0 = (q_0, q_{0,\mathcal{M}})$ is the initial state,
- $\Sigma_I = L$ is the input alphabet,
- $\Sigma_O = A$ is the output alphabet,
- δ is the next-state function with

$$\delta((q, q_{\mathcal{M}}), l, a) = (\delta(q, (l, a)),\ \delta_{\mathcal{M}}(q_{\mathcal{M}}, (l, a))),$$

  for all $(q, q_{\mathcal{M}}) \in G$, $l \in L$, $a \in A$, and

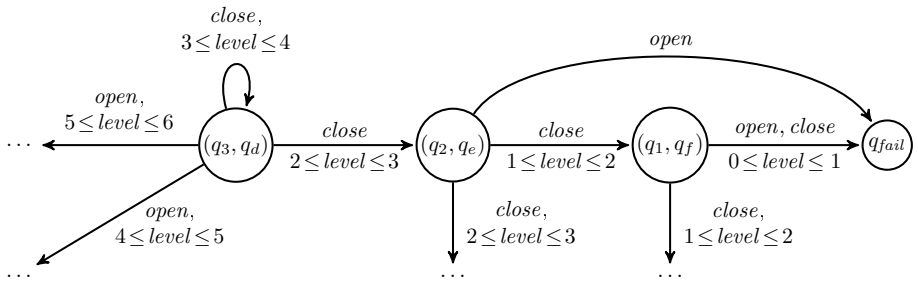


Figure 6.7: An excerpt for the product game of the water storage tank example.

- F^g is the set of safe states with

$$F^g = (F \times Q_{\mathcal{M}}) \cup (Q \times (Q_{\mathcal{M}} \setminus F_{\mathcal{M}})).$$

In the construction, the state space of the game is the product between the specification automaton state set and the abstraction state set. The safe states in the game (in the set F^g) are the ones at which either the specification automaton is in a safe state, or the abstraction is in an unsafe state. The latter case represents that the observed MDP behavior differs from the behavior that was modeled in the abstraction. For game solving, it is important that such cases (whose occurrence in the field witnesses the incorrectness of the abstraction) count as winning for the system player, as the system player only needs to play correctly in environments that conform to the abstraction.

Step 2. Compute the winning region W of $\mathcal{G}$: In the second step, we compute the winning region $W \subseteq F^g$ of G by standard safety game solving as described by Bloem et al. [BKKW15].

Example 8

To exemplify the shield construction, let us reconsider Example 7. Building the product game between the specification automaton φ^s and the MDP abstraction $\varphi^{\mathcal{M}}$ leads to a safety game with 602 states (if we merge all states in $(Q \setminus F) \times Q_{\mathcal{M}}$ into a single error state and all states in $Q \times (Q_{\mathcal{M}} \setminus F_{\mathcal{M}})$ into a single paradise state *from which the game is always won by the system). If we solve the game, then most of the states are winning, but a few are not. Figure 6.7 shows a small fraction of the game that contains such non-winning states. Transitions to paradise states are not shown. In state (q_3, q_d), the system should not choose action close, as otherwise the system cannot avoid to reach q_{fail} for some possible evolution of the environment that is consistent with our abstraction. It could be the case that q_{fail} is not reached when the environment chooses to let the level stay the same for a step, but the system cannot be sure about this, so the action close must not be picked.*

A shield allows all actions that are guaranteed to lead to a state in W, no matter what the next observation 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 error state (according to the abstraction).

Step 3. Translate $\mathcal{G}$ and W to a reactive system $\mathcal{S}$: We translate $\mathcal{G}$ and W to a reactive system $\mathcal{S} = (Q_\mathcal{S}, q_{0,\mathcal{S}}, \Sigma_{I,\mathcal{S}}, \Sigma_{O,\mathcal{S}}, \delta_\mathcal{S}, \lambda_\mathcal{S})$ that constitutes the post-posed shield. The shield has the following components:

- $Q_\mathcal{S} = G$ is the state space,
- $q_{0,\mathcal{S}} = g_0$ is the initial state,
- $\Sigma_{I,\mathcal{S}} = L \times A$ is the input alphabet,
- $\Sigma_{O,\mathcal{S}} = A$ is the output alphabet,
- $\delta_\mathcal{S}$ is the next-state function with

  $$\delta_\mathcal{S}(g, l, a) = \delta(g, l, \lambda_\mathcal{S}(g, l, a))$$

  for all $g \in G, l \in L, a \in A$, and
- $\lambda_\mathcal{S}$ is the output function with

$$\lambda_\mathcal{S}(g, l, a) = \begin{cases} a & \text{if } \delta(g, (l, a)) \in W \\ a' & \text{if } \delta(g, (l, a)) \notin W \text{ for some arbitrary} \\ & \text{but fixed } a' \text{ with } \delta(g, (l, a')) \in W. \end{cases}$$

The construction of a shield can be extended naturally if a ranking of actions $rank_t = \{a_t^1, \ldots, a_t^n\}$ is provided by the agent. Then, the shield selects the first action $a_t = a_t^i$ that is allowed by φ^s. Only if all actions in $rank_t$ are unsafe, the shield is allowed to deviate and to select a safe action $a_t \notin rank_t$.

Theorem 6
A reactive system $\mathcal{S}$ constructed according to Algorithm 7 is a post-posed shield (Definition 3).

Proof. We have to proof that $\mathcal{S}$ has the claimed properties, namely *correctness* and *minimal interference*.

Correctness: A shield works correctly if for every trace $s_0 a_0 s_1 a_1 \ldots \in (S \times A)^\omega$ that MDP, shield and learner can together produce, we have that

$$\overline{\sigma} = (f(s_0), a_0)(f(s_1), a_1) \ldots \vDash \varphi^S.$$

Let $\mathcal{S}(\overline{\sigma}) = q_{\mathcal{S},0} q_{\mathcal{S},1} \ldots \in Q_\mathcal{S}^\omega$ be the run of the shield, with $q_{\mathcal{S},i} = (q_i, q_{\mathcal{M},i})$. for all i. Therefore, $q_{\mathcal{M},0} q_{\mathcal{M},1} \ldots$ is the run of $\varphi^\mathcal{M}$ and $q_0 q_1 \ldots$ is a run of φ^S.

$\lambda_{\mathcal{S}}$ of $\mathcal{S}$ corrects all actions that would lead to states outside of the winning region. Therefore, $\mathcal{S}$ only has reachable states $q_{\mathcal{S}} = (q, q_{\mathcal{M}})$ that are in the set of winning positions, i.e., for all states $q_{\mathcal{S}} \in Q_{\mathcal{S}}$, and all possible next labels $l \in L$, there exists at least one action $a \in A$ such that if a is taken, then the next state $q' = (q', q'_{\mathcal{M}})$ is winning as well. Since only winning states are visited during a play, the error state of φ^S can only be visited after the error state of $\varphi^{\mathcal{M}}$ has been visited (and hence the abstraction turned out to be incorrect). Hence, $\mathcal{S}$ ensures correctness.

Minimal Interference: Let the shield, learner, and MDP together produce a trace $s_0 a_0 s_1 a_1 s_2 a_2 \dots s_k$. Assume that $\mathcal{S}$ overwrites an action a_{k+1} that is available from state s_k in the MDP.

We show that $\mathcal{S}$ had to overwrite a_{k+1} as there is another MDP that is consistent with the observed behavior and the abstraction for which, regardless of the learner's policy, there is a non-zero probability to violate the specification after the trace $s_0 a_0 s_1 a_1 s_2 a_2 \dots s_k a_{k+1}$. Using $\varphi^{\mathcal{M}}$, we define this other MDP $\mathcal{M}' = (S', s'_I, A, P')$ with the following components:

- $S' = Q_{\mathcal{M}} \times L$ is the state space,
- $s'_I = (q_{\mathcal{M},0}, f(s_0))$ is the initial state, and
- the probabilistic transition function $P'((q_{\mathcal{M}}, l), a)$ is a uniform distribution over all elements from the set $\{(q'_{\mathcal{M}}, l') \in Q_{\mathcal{M}} \times L \mid q'_{\mathcal{M}} = \delta_{\mathcal{M}}(q_{\mathcal{M}}, (l, a)), q'_{\mathcal{M}} \in F_{\mathcal{M}}, \exists a' \in A. \delta_{\mathcal{M}}(q'_{\mathcal{M}}, (l', a')) \in F_{\mathcal{M}}\}$ for every $(q_{\mathcal{M}}, l) \in S'$ and $a \in A$,

Assume now, that an action a_{k+1} is overwritten by another action from $\mathcal{S}$ after the trace $s_0 a_0 s_1 a_1 s_2 a_2 \dots s_k$ while $\mathcal{S}$ is in a state $(q, q_{\mathcal{M}})$. We have that $\mathcal{M}'$ is an MDP in which every finite-length label sequence that is possible in the abstraction for some action sequence has a non-zero probability to occur if the action sequence is chosen. Due to the construction of $\mathcal{S}$, action a_{k+1} is only forbidden in state $(q, q_{\mathcal{M}})$ if in the game, the environment player had a strategy to violate φ^S using only traces allowed $\varphi^{\mathcal{M}}$. Since φ^S is a safety property, the violation would occur in finite time. Since in $\mathcal{M}'$, all finite traces that can occur in $\varphi^{\mathcal{M}}$ have a non-zero probability, allowing a_{k+1} would imply a non-zero probability to violate φ^S in the future. Hence, $\mathcal{S}$ could not prevent a violation in such a case, and a_{k+1} needs to be corrected. Hence, $\mathcal{S}$ is minimal interfering. **Q. E. D.**

6.4 Preemptive Shields for RL

6.4.1 Preemptive Shielding Setting for RL

Figure 6.8 depicts the preemptive shielding learning setting. The interaction between the agent, the environment and the shield is as follows. At every time step t, the shield computes a set of all safe actions $\{a_t^1, \dots, a_t^k\}$, i.e., it takes the set of all actions available, and removes all unsafe actions that would violate the

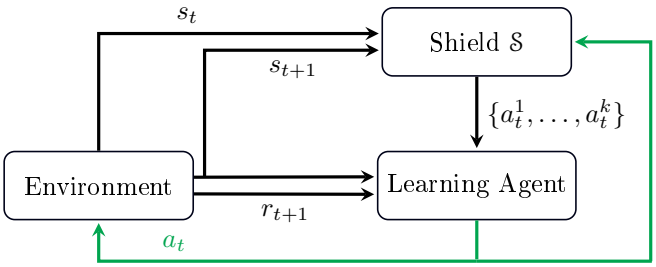


Figure 6.8: Preemptive shielded reinforcement learning - Detail

safety specification φ_s. The agent receives this list from the shield, and picks an action $a_t \in \{a_t^1, \ldots, a_t^k\}$ from it. The environment executes action a_t, moves to a next state s_{t+1}, and provides the reward r_{t+1}. The task of the shield is basically to modify the set of available actions of the agent in every time step such that only safe actions remain.

For a preemptive shield, we have $\Sigma_O = 2^A$, as the shield outputs the set of actions for the learner to choose from for the respective next step. The shield observes the label of the last MDP state in the sequence so far and provides the set of safe actions. For selecting the next transition of the finite-state machine that represents the shield, it also makes use of the action actually chosen by the agent. So for the input alphabet of the shield, we have $\Sigma_I = L \times A \times L$. See Section 3.3 for a more detailed explanation of the input and output alphabet of preemptive shields.

The shield and the learner together produce a trace $s_0 a_0 s_1 a_1 \ldots \in (S \times A)^\omega$ in the MDP $\mathcal{M} = \{S, s_i, A, P, R\}$ if there exists a trace $q_0 q_1 \ldots \in Q^\omega$ in the shield such that, for every $i \in \mathbb{N}$, we have $a_i \in \lambda(q_i, L(s_i))$ and $q_{i+1} = \delta(q_i, (L(s_i), a_i)$.

Properties of Preemptive Shielding. The preemptive shielding approach can also be seen as transforming the original MDP $\mathcal{M} = \{S, s_i, A, P, R\}$ into a new MDP $\mathcal{M}' = (S', s_I, A', P', R')$ with the unsafe actions at each state removed, and where S' is the product of the original MDP and the state space of the shield. For each $s \in S'$, we create a new subset of available actions $A'_s \subseteq A_s$ by applying the shield to A_s and eliminating all unsafe actions. From each state $s \in S'$, the transition function P' contains only transition distributions from P for actions contained in A'_s.

6.4.2 Synthesis of Preemptive Shields for RL

Algorithm 8 — Synthesis of Preemptive Shields for RL
Starting from $\varphi^s = (Q, L \times A, \Sigma, \delta, F)$ *and* $\varphi^{\mathcal{M}} = (Q_{\mathcal{M}}, q_{0,\mathcal{M}}, A \times L, \delta_{\mathcal{M}}, F_{\mathcal{M}})$*, to compute a preemptive shield, we perform the following steps:*

Steps 1-2. Perform as in Section 6.3.2.

Step 3. Translate $\mathcal{G}$ and W to a reactive system $\mathcal{S}$: We translate $\mathcal{G}$ and W to a reactive system $\mathcal{S} = (Q_\mathcal{S}, q_{0,\mathcal{S}}, \Sigma_{I,\mathcal{S}}, \Sigma_{O,\mathcal{S}}, \delta_\mathcal{S}, \lambda_\mathcal{S})$ that constitutes the post-posed shield. The shield has the following components:

- $Q_\mathcal{S} = G$ is the state space,
- $q_{0,\mathcal{S}} = g_0$ is the initial state,
- $\Sigma_{I,\mathcal{S}} = L \times A \times L$ is the input alphabet,
- $\Sigma_{O,\mathcal{S}} = 2^A$ is the output alphabet,
- $\delta_\mathcal{S}$ is the next-state function with

  $$\delta_\mathcal{S}(g, l, a) = \delta(g, l, a)$$

  for all $g \in G, l \in L, a \in A$, and
- $\lambda_\mathcal{S}$ is the output function with

  $$\lambda_\mathcal{S}(g, l, a, l') = \{a' \in A \mid \delta(\delta(g, l, a), l', a') \in W\}$$

  for all $g \in G$, $a \in A$, and $l \in L$.

For selecting the next transition, the preemptive shield makes use of the action actually chosen by the agent, i.e., the next transition is chosen via the last observed state from the MDP l and the last executed action a. The next output of the shield contains all actions a', that lead from the current state of the shield $g' = \delta(g, l, a)$ and the current observation of the MDP l' to another state in the winning region, i.e., a' is correct.

Theorem 7
A reactive system $\mathcal{S}$ constructed according to Algorithm 8 is a preemptive shield (Definition 18).

Proof. We have to proof that $\mathcal{S}$ is correct and minimal interfering. For both, the proof is very similar to the proof for post-posed shields discussed in Detail in Section 6.3.2.

Correctness: $\lambda_\mathcal{S}$ of $\mathcal{S}$ deactivates all actions that would lead to states outside of the winning region. Therefore, only winning states are ever visited along a play, i.e., the error state of φ^S can only be visited after the error state of $\varphi^\mathcal{M}$ has been visited. Hence, $\mathcal{S}$ ensures correctness.

Minimal Interference: We construct an MDP $\mathcal{M}' = (S', s'_I, A, P')$ in which every finite-length label sequence that is possible in $\varphi^{\mathcal{M}}$ for some action sequence has a non-zero probability (see Section 6.3.2 for the construction of $\mathcal{M}'$).

Assume, that $\mathcal{S}$ deactivates a_{k+1} after the trace $s_0 a_0 s_1 a_1 s_2 a_2 \dots s_k$, while the shield is in a state $(q, q_{\mathcal{M}})$. Due to the construction of $\mathcal{S}$, action a_{k+1} is only deactivated in $(q, q_{\mathcal{M}})$ if the environment player had a strategy to violate φ^S using only traces allowed by $\varphi^{\mathcal{M}}$. Since in $\mathcal{M}'$, all finite traces that can occur in $\varphi^{\mathcal{M}}$ have a non-zero probability, allowing a_{k+1} would imply a non-zero probability to violate φ^S in the future, for any action sequence. Hence, a_{k+1} needs to be deactivated and $\mathcal{S}$ is minimal interfering. **Q. E. D.**

6.5 Convergence

Define an MDP $\mathcal{M} = (S, s_I, A, P, R)$, with discrete state set S, discrete state-dependent action sets A, and state-dependent transition functions $P(s, a, s')$ that define the probability of transitioning to state s' when taking action a in state s. Assume also that a shield $\mathcal{S} = (Q_{\mathcal{S}}, q_{\mathcal{S},0}, \Sigma_{I,\mathcal{S}}, \Sigma_{O,\mathcal{S}}, \delta_{\mathcal{S}}, \lambda_{\mathcal{S}})$ is given for $\mathcal{M}$ and for some MDP labeling function $f : Q \to L$.

We can build a product MDP $\mathcal{M}'$ that represents the behavior of the shield and the MDP together. Since $\mathcal{M}'$ is a standard MDP, all learning algorithms that converge on standard MDPs can be shown to converge in the presence of a shield under this construction.

Note, that for the post-posed shielding case, this argument requires that whenever an unsafe action is chosen by the learner, there is a fixed probability distribution over the safe actions executed instead. This distribution may depend on the state of the MDP and the shield and the selected ranking, but must be constant over time, as otherwise we could not model the joint behavior of the shield and the environment MDP as a product MDP.

6.6 Experimental Results

We applied post-posed shielded RL in two domains: (1) the water tank scenario from Example 7, and (2) a simple PacMan example. The simulations were performed on a computer equipped with an Intel® Core™i7-4790K and 16 GB of RAM running a 64-bit version of Ubuntu® 16.04 LTS.

6.6.1 A Shield for a Water Tank

In this example, the tank must never run dry or overflow by controlling the inflow switch (φ_1^s). In addition, the inflow switch must not change its mode of operation before three time steps have passed since the last mode change (φ_2^s). Refer to Example 7, for a full description of the abstract water tank dynamics and specification. We generated a concrete MDP in which the energy consumption depends only on the state, and there are multiple local minima. A

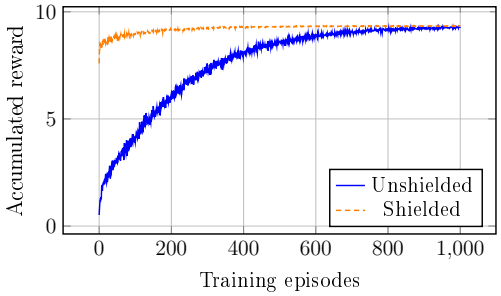


Figure 6.9: Accumulated reward for the water tank example.

post-posed shield was synthesized in less than a second. Figure 6.9 shows that the shielded (orange dashed line) and unshielded (blue solid line) Q-learning experiment do reach an optimal policy. However, the shielded implementation reaches the optimal policy in a significantly shorter time than the unshielded implementation.

6.6.2 A Shield for simple PacMan

To demonstrate our methods, we conducted another experiment on the arcade game PacMan. The task is to eat food in a maze and not get eaten by the *ghost*. PacMan achieves a high score if it eats all the food as quickly as possible while minimizing the number of times to get eaten by the ghosts. RL approaches exist [Ber18], but they suffer primarily from the fact that during the exploration phase, PacMan is eaten by the ghosts many times and thus achieves very poor scores.

Setting. We employ a script that can automatically generate an abstraction $\varphi^{\mathcal{M}}$ for any maze. $\varphi^{\mathcal{M}}$ encodes all possible movements of PacMan and the ghost. The safety specification φ^{S} states that PacMan should *not be eaten* by the ghost. During the play, PacMan achieves the following scores: food earns reward (+10), while each step causes a small penalty (−1). A large reward (+500) is granted if PacMan eats all the food in the maze. If PacMan gets eaten, a large penalty (−500) is imposed, and the game is restarted.

Reinforcement Learning. Our implementation employs an existing PacMan environment[1] and uses an approximate Q-learning agent [SB98] with the following feature vector: (1) how far away the next food is, (2) whether a ghost collision is imminent, and (3) whether a ghost is one step away. The result is a basic reflex controller for PacMan. The Q-learning uses the learning rate $\alpha = 0.2$, the discount factor $\gamma = 0.8$ for the Q-update, and an ϵ-greedy exploration policy with $\epsilon = 0.05$.

[1] `http://ai.berkeley.edu/project_overview.html`

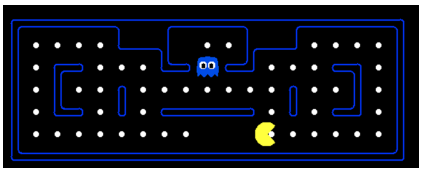


Figure 6.10: Simple PacMan

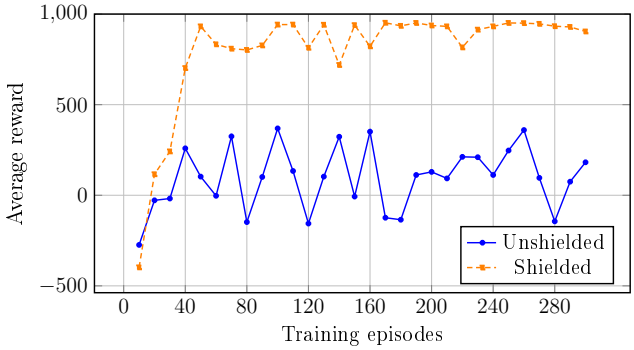


Figure 6.11: Resulting scores for simple PacMan

Results. We consider the labyrinth illustrated in Figure 6.10 with one ghost. We synthesized a post-posed shield in less than seven seconds. We compare RL to shielded RL. The key comparison criterion is the *score* composed by rewards and penalties, as explained above. Figure 6.11 shows the scores obtained during RL. In particular, the curves (blue, solid: unshielded, orange, dashed: shielded) show the average scores for every ten training episodes. One episode lasts until either the game is won (all food is eaten) or lost (ghost eats PacMan). As expected, in the shielded case, PacMan is always able to clear the game starting from the first episode, since the shield always saves it. Moreover, the shield helps to learn an optimal policy much faster because fewer restarts are needed.

Declaration of Sources

Chapter 6 was based on and reuses material from the following sources, previously published by the author:

- [ABE+17] M. Alshiekh, R. Bloem, R. Ehlers, B. Könighofer, Scott Niekum, and Ufuk Topcu. *Safe reinforcement learning via shielding.* arXiv. 2017
- [ABE+18] M. Alshiekh, R. Bloem, R. Ehlers, B. Könighofer, Scott Niekum, and Ufuk Topcu: *Safe reinforcement learning via shielding.* AAAI. 2018

References to these sources are not always made explicit.

7

Safe Reinforcement Learning via Probabilistic Shields

7.1 Motivation and Outline

In Chapter 6, we discussed deterministic shields as a suitable technique to guarantee correctness with certainty in reinforcement learning. Deterministic shields prevent an agent from taking any unsafe actions at runtime. To this end, the performance objective of the learning agent is extended with a constraint specifying that unsafe states are *never* visited, i.e., there are no safety violations (the safety objective).

However, in many cases, this tight restriction on the decision-making of the agent to adhere to the safety requirements limits the agent's exploration and understanding of the environment [PS14, AOS+16]. Such restrictions may cause *insufficient progress* in following the original objective of the decision-maker, or policies satisfying the restrictions may not exist.Thus, there is a *trade-off between safety* and *progress*.

Example 9
For instance, take a self-driving car. The main objective of the underlying AI is to follow a road while avoiding accidents. The objective of the shield is to prevent accidents by enforcing emergency brakes. However, there is the tradeoff mentioned above between safety and progress: In critical situations, the car has to brake, but frequent (unnecessary) emergency brakes are not desirable in terms of many performance measures. To avoid too many interferences by the shield, the shield has to evaluate how critical the current situation is. Therefore, the shield must take the inherent randomness and uncertainty in the actions of other

traffic participants into account.

In this chapter, we propose *probabilistic shields* that incorporate more liberal constraints that enforce safety violations to occur *only with a small probability.* **If an action increases the probability of a safety violation by more than a factor $1/\delta$ with respect to the optimal safety probability, the shield blocks the action from the agent.**

Consequently, an agent augmented with a shield is *guided* to satisfy the safety objective during exploration (or as long as the shield is used). The shield is *adaptive* with respect to δ, as a high value for δ yields a stricter shield, a smaller value a more permissive shield. The value for δ can be changed on-the-fly and may depend on the individual minimal safety probabilities at each state.

We base our formal notion of a probabilistic shield on Markov decision processes (MDPs), which constitute a popular modeling formalism for decision-making under uncertainty [Whi85] and is widely used in model-based RL. We assess safety using probabilistic temporal logic constraints [BK08] that limit, for example, the probability for reaching a set of critical states in the MDP.

Advantages of probabilistic shields. We identify the following key benefits for shielding MDPs:

- **Scalability.** Model checking—as any model-based technique—is susceptible to scalability issues. A key advantage of using a separate safety objective is that we may analyze safety on just a fraction of the system. In our experiments, these MDP fragments are at least ten orders of magnitude smaller than a full model of the system, rendering model checking applicable to realistic scenarios and enables the usage of model checking to compute a shield. Note that the learner respecting the performance has to consider the full model, but may do so using either model-based or model-free approaches.

- **Adaptivity.** Without randomness, all states are either absolutely safe or unsafe. However, in the presence of randomness, safety may be seen as a quantitative measure: in some states, all actions may induce a large risk, while one action may be considered *relatively* safe. Therefore, it is essential to have an *adaptive* notion of shielding, in which the pre-selection of actions is not based on absolute thresholds, i.e., if necessary, the shield needs to dynamically adapt to allow more, potentially less safe, decisions.

- **Trade-off between safety and progress.** Shielding may *restrict* exploration and lead to suboptimal policies. Therefore, it should not be considered in isolation. The aforementioned trade-off between optimizing the performance objective and the achieved safety is intricate. Intuitively, taking a bit additional risk short-term may allow for efficient exploration and limit the risk long-term. To this end, we provide and discuss mechanisms that allow adjusting the shield based on such observations.

Construction of probabilistic shields. We outline the synthesis process to create probabilistic shields (see Section 7.3 for more details).

1. For a large MDP setting with a potentially unknown reward function, we construct a smaller MDP that is sufficient for safety assessments, called the *safety-relevant MDP*. In particular, we first construct a behavior model for the environment, for example using model-based RL [DN08], in a training environment together with suitable data augmentation techniques. We can plug this behavior model into any concrete scenario to obtain the safety-relevant MDP for the shield computation. While reinforcement learning targets the full scenario, we never need the full (large) MDP for the construction of the shield.

2. The shield is computed from the safety-relevant MDP and the safety specification. For any state and any possible decision, the synthesis procedure computes precise probabilities for violating the safety specifications. Based on these values and the factor δ, the shield deactivates actions that induce a too large risk.

3. The probabilistic shield then readily augments either model-free or model-based RL and is placed before the learning agent. At each time step, the shield provides a list of actions with the actions that are too risky deactivated.

Outline. In the remainder of this chapter, we first describe the setting and the problem that we want to solve in Section 7.2. Next, we describe in detail how to construct a probabilistic shield and provide optimizations towards a computationally tractable implementation in Section 7.3. We demonstrate several concepts on how to maintain sufficient progress in exploring an environment while shielding decision-making. In Section 7.4, we demonstrate our implementation and experiments based on the arcade game PacMan and a case study involving service robots in a warehouse. Shielded RL leads to improved policies for both case studies with fewer safety violations and performance superior to unshielded RL.

7.2 Probabilistic Shielding Setting

In this section, we first define the setting in which we want to apply the probabilistic shield and discuss several potential applications. Next, we give a problem statement about what we would like to achieve by applying a shield in the discussed setting.

We define a partially-controlled multi-agent system [BT96, VdHW08], where one controllable agent (the *avatar*) and a number of uncontrollable agents (the *adversaries*) operate within a finite graph representation of an arena. The arena is a compact, high-level description of the underlying model. From this arena,

the potential states and actions of all agents may be inferred. For safety considerations, the reward structure can be neglected, effectively reducing the state space for our model-based safety computations.

Definition 21 — Arena
An arena *is a directed graph* $\mathcal{A} = (V, E)$ *with a finite sets* V *of nodes and* $E \subseteq V \times V$ *of edges. The agent's* position *is defined via the current node* $v \in V$*. The agent* decides *on a new edge* $(v, v') \in E$ *and determines its next position* v'*.*

Some combinations of agent positions are safety-critical (e.g., they may correspond to collisions). A safety property may describe reaching such positions, or any other property expressible in the safety fragment of temporal logic.

While the underlying model for the arena suffices to specify the safe behavior, it is not sufficiently succinct to model the performance via rewards. Consider an edge that is safety-relevant, but the agent is only rewarded the first time taking this edge. Thus, in a flat model with rewards, two different edges are necessary to model this behavior. However, the reward (and thus the difference between these edges) is not needed to assess the safety, and the safety-relevant model may be pruned to an exponentially smaller model. We use a *token* function that implicitly extends the underlying model by a reward structure, enabling separation of concerns between safety and performance.

Technically, we associate edges with a token function $\circ\colon E \to \{0, 1\}$, indicating the status of an edge. Naturally, tokens can be extended to describe an n-ary status. Tokens can be (de-) activated and have an associated *reward* earned upon taking edges with an active token. The performance objective is the maximization of the expected reward.

Application. In the experiments for this chapter, we demonstrate our approach using the arcade game PacMan and a case study involving service robots in a warehouse. However, we designed the formal setting to be applicable to a series of further scenarios. We detail a few possibilities here that go beyond other arcade games.

- **Autonomous driving.** An autonomous taxi (the avatar) operates within a road network encoded by an arena. The taxi has to visit several points to pick up or drop off passengers. Upon visiting such a point, a corresponding token activates, and a reward is earned. Afterward, the token is deactivated permanently. Meanwhile, the taxi has to account for other traffic participants or further environmental factors (the adversaries) [Die00], for which we may learn behavior models over time [SSSD16, SLS+18]. A sensible safety specification may restrict the probability for collision with other cars, say, to 0.05. Note that the token structure is not relevant for such a specification. By employing our shielding technique, we can achieve *safe* learning to obtain an optimal route for the taxi.

- **Robot logistics in a smart factory.** Take a factory floor plan with several corridors. The nodes describe crossings, the edges the corridors

with the machines, and the distances the lengths of the corridors. The adversaries are (possibly autonomous) transporters moving parts within the factory. The avatar models a specific service unit, moving around and inspecting machines where an issue has been raised (as indicated by a token), while accounting for the behavior of the adversaries. All agents follow the corridors and take another corridor upon reaching a crossing. Corridors might be to narrow for multiple (facing) robots, which poses a safety-critical situation. The tokens allow having a state-dependent cost, either as long as they are present (indicating the costs of a broken machine) or for removing the tokens (indicating costs for inspecting the machine). A similar scenario has been investigated in [BLP+18].

Problem. Consider an environment described by an arena as above and a safety specification. We assume stochastic behaviors for the adversaries, e.g, obtained using RL [SLS+18, SSSD16] in a training environment. In fact, this stochastic behavior determines all actions of the adversaries via probabilities. The underlying model is then a Markov decision process: the avatar executes an action, and upon this execution the next exact positions (the state of the system) are determined stochastically.

We compute a δ-shield that prevents avatar decisions that violate this specification by more than a threshold δ with respect to the optimal safety probability. We evaluate the shield using a model-based or model-free RL avatar that aims to optimize the performance. The shield therefore has to handle an intricate trade-off between strictly focussing on (short and midterm) safety and performance.

7.3 Synthesis of Probabilistic Shields

We outline the workflow of our approach in Figure 7.1 and below, starting with the setting from the previous section. We employ a separation of concerns between the model-based shield construction and potentially model-free reinforcement learning (RL). First, based on observations in multiple arenas, we construct a general (stochastic) *behavior model* for each adversary. Combining these models with a concrete arena yields to a compact MDP model. At this point, we ignore the token function and the potentially unknown reward function, so the MDP is a safety-relevant quotient of the full model that models the real system within which we only assess safe behavior. We therefore call the MDP the safety-relevant quotient. The underlying *full MDP* incorporates the *token function*, where associated rewards may be known or observed during learning. Including tokens constitutes an exponential blowup of the safety-relevant quotient, rendering probabilistic model checking or planning practically infeasible. Using the safety-relevant MDP, we construct a *shield* using probabilistic model checking.

We now detail the individual technical steps to realize our proposed method.

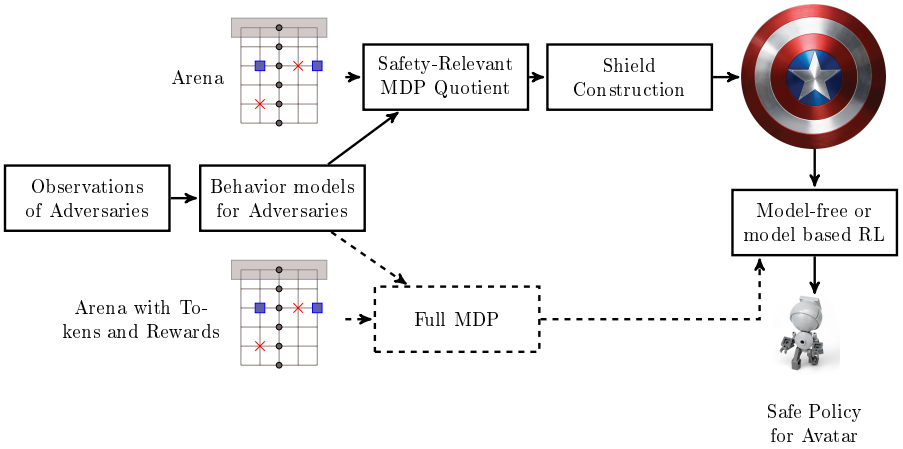


Figure 7.1: Workflow of the Shield Construction

Step 1. Construct behavior models for adversaries: We learn an adversary model by observing behavior in a set of similar (small) arenas, until we gain sufficient confidence that more training data would not change the behavior significantly [SLS+18, SSSD16]. An upper bound on the necessary data may be obtained using Hoeffding's inequality [ZMBD08]. To reduce the size of the training set, we devise a data augmentation technique using domain knowledge of the arenas [KSH12, WFHP16]. In particular, we abstract away from the precise configuration of the arena by partitioning the graph into zones that are relative to the view-point of the adversary (e. g., near or far, north or south, east or west). The intuitive assumption is that the specific position of an adversary is not important, but some key information is (e.g., the relation to the position of the avatar). This approach (1) speeds up the learning process and (2) renders the resulting behavior model applicable for varying the concrete instance of the same setting.

Zones are uniquely identified by a coloring with a finite set C of colors.

Definition 22 — Zones
For an arena $G = (V, E, d)$, zones relative to a node $v \in V$ are given by a function $z_v \colon V \to C$, where C is a finite set of colors.

For nodes $x, y \in V$, with $z_v(x) = z_v(y)$, the assumption is that the adversary in v behaves similarly regardless whether the avatar is in x or y. From our observations, we extract a *histogram* $h \colon E \times C \to \mathbb{N}$, where $h(e, c)$ describes how often the adversary takes an edge $e = (v, v') \in E$ while the avatar is in a node u with $z_v(u) = c$. We translate these likelihoods into distributions over the possible edges in the arena to obtain the behavior model for adversaries.

Definition 23 — Adversary Behavior
For an arena $G = (V, E, d)$, zones $z_u \colon V \to C$ for every $u \in V$, and a histogram

$h\colon E \times C \to \mathbb{N}$, *the* adversary behavior *is a function* $B\colon V \times C \to \mathit{Distr}(E)$ *with*

$$B(v,c) = \frac{h\big((v,v'),c\big)}{\sum_{(v,v')\in E} h\big((v,v'),c\big)} \ .$$

While we employ a simple normalization of likelihoods, alternatively one may also utilize, e. g., a softmax function which is adjustable to favor more or less likely decisions [SB98].

Step 2. Compute the safety-relevant quotient MDP: The construction of the safety-relevant (quotient) MDP $\mathcal{M} = (S, A, P)$ augments an arena by behavior models B_i for adversaries.

Starting from an arena $\mathcal{A} = (V, E)$ with agents $0, \ldots, m$, where agent 0 is the avatar and agents $1, \ldots, m$ are adversaries, and the adversary behavior models $B_1, \ldots, B_m$, we compute the *safety-relevant quotient MDP* $\mathcal{M} = (S, A, P)$, such that:

- $S = V^{m+1} \times \{0, \ldots, m\}$ is the state space,
- $A = \{a_0\} \cup A_E$ with $A_E = \{a_e \mid e \in E\}$ is set of actions, and
- P is the probabilistic transition function with

$$P(s,a,s') = \begin{cases} 1 & \text{if } \ s = (v_0,\ldots,v_m,0), s' = (v_0',v_1\ldots,v_m,1), \\ & \ a = a_e \text{ and } (v_0,v_0') = e \in E \\ B_i(v_i,c)(v_i,v_i') & \text{if } \ i \neq 0, s = (v_0,\ldots,v_m,i), \\ & \ s' = (v_0',\ldots,v_m',(i+1) \bmod m), \\ & \ (v_i,v_i') = e \in E, v_j' = v_j \text{ for } j \neq i, \\ & \ a = a_0 \text{ and } z_v(v_0) = c. \end{cases}$$

The reward function r remains undefined.

The *states* $S = V^{m+1} \times \{0, \ldots, m\}$ encode the positions for all agents and whose turn it is. The *actions* $A = \{\alpha_0\} \cup A_E$ with $A_E = \{\alpha_e \mid e \in E\}$ determine the movements of the avatar and the adversaries. Observe there are only certain states where the avatar has to make choices.

Definition 24 — Decision States
The decision states *of the safety-relevant MDP* $\mathcal{M} = (S, A, P)$ *are* $S_d = \{s_d \in S \mid s_d = (\ldots, 0)\}$*, i. e., it's the turn of the avatar.*

A policy only needs to select actions at the decision states as at all other states only the unique action a_0 emanates. Consequently, a policy for $\mathcal{M}$ is a policy for the avatar.

For $(v, \ldots, 0) = s_d \in S_d$ (the avatar moves next), the available actions are $a_e \in A(s_d) \subseteq A_e$, where a_e corresponds to an outgoing edge of v. For $(v, \ldots, 0) = s_d \in S_d$, a_e with $e = (v, v')$ leads with probability one to a state $s_e = (v', \ldots, 1)$.

For $(v, \ldots, v_i, \ldots, i > 0)$ (an adversary moves next), there is a unique action a_0 where v_i is changed to v_i', randomly determined according to the behavior B_i, which also updates i to $i+1$ modulo m. These transitions induce the only probabilistic choices in the MDP.

In theory, one can build the full MDP for the arena (V, E) and the token function $\circ\colon E \to \{0,1\}$ under the assumption that the reward function is known. Then, one can compute the reward-optimal and safe policy without need for further learning techniques. As there are 2^E token configurations, the state space blows up exponentially, which prevents the successful application of model checking or planning techniques for anything but very small applications.

Step 3. Construct the δ-shield: For an arena $\mathcal{A} = (V, E)$ and the corresponding safety-relevant MDP $\mathcal{M} = (S, A, P)$, a set of unsafe states $T \subseteq S$ should preferably not be reached from any state. The property $\varphi = \Diamond T$ encodes the *violation* of this safety constraint, that is, eventually reaching T within $\mathcal{M}$. The shield needs to limit the probability to *satisfy* φ. We evaluate all decision states $s_d \in S_d$ with respect to this probability: We compute $\eta^{\min}_{\varphi,\mathcal{M}}(s_e)$, i.e., the minimal probability to satisfy φ from s_e, which is the state reached after taking action $a_e \in A_e$ in s_d [CHVB18].

Definition 25 — Action-valuation and optimal action-value
An action-valuation *for action $a_e \in A_e$ at state $s_d \in S_d$ is*

$$val^{\mathcal{M}}_{s_d}\colon A(s_d) \to [0,1], \text{ with } val^{\mathcal{M}}_{s_d}(a_e) = \eta^{\min}_{\varphi,\mathcal{M}}(s_e)\ .$$

The set of all action-valuations at s_d is $ActVals_{s_d}$. The optimal action-value *for s_d is*

$$optval^{\mathcal{M}}_{s_d} = \min_{a' \in A} val^{\mathcal{M}}_{s_d}(a').$$

We now define a shield for the safety-relevant MDP $\mathcal{M}$ using the action values. Specifically, a *δ-shield* for $\delta \in [0,1]$ determines a set of actions at each decision state s_d that are δ-optimal for the specification φ. All other actions are "shielded" or "blocked".

Definition 26 — δ-Shield
For action-valuation $val^{\mathcal{M}}_{s_d}$ and $\delta \in [0,1]$, a δ-shield for state $s_d \in S_d$ *is*

$$\mathcal{S}^{s_d}_{\delta}\colon ActVals_{s_d} \to 2^{A(s_d)}$$

with $\mathcal{S}^{s_d}_{\delta} \mapsto \{a \in A(s_d) \mid \delta \cdot val^{\mathcal{M}}_{s_d}(a) \leq optval^{\mathcal{M}}_{s_d}\}$.

Intuitively, δ enforces a constraint on actions that are acceptable with respect to the optimal probability. The shield is *adaptive* with respect to δ, as a high value for δ yields a stricter shield, a smaller value a more permissive shield. The shield is stored using a lookup-table, and the value for δ can then be changed on-the-fly. In particularly critical situations, the shield can enforce the decision-maker to resort to (only) the optimal actions with respect to the safety objective.

A δ-shield for the MDP $\mathcal{M}$ is built by constructing and applying δ-shields to all decision states.

Definition 27 — Shielded MDP
The shielded MDP $\mathcal{M}_{\text{U}} = (S, A\, P_{\text{U}})$ *for a safety-relevant quotient MDP* $\mathcal{M} = (S, A, P)$ *and a* δ*-shield for all* $s_d \in S_d$ *is given by the transition probability* P_{U} *with*

$$P_{\text{U}}(s,a) = \begin{cases} P(s,a) & \text{if } a \in \mathcal{S}^s_\delta(val^{\mathcal{M}}_s) \\ \bot & \text{otherwise.} \end{cases}$$

At least the actions that induce the optimal probability to satisfy φ are always present in the shielded MDP. There is no state where all actions are blocked by the shield.

Lemma 1
The MDP $\mathcal{M}$ *is deadlock-free if and only if the shielded MDP* $\mathcal{M}_{\text{U}}$ *is deadlock-free.*

We compute the shield relative to optimal values $optval^{\mathcal{M}}_{s_d}$. Consequently, for $\delta = 1$, only optimal actions are preserved, and for $\delta = 0$ no actions are blocked.

Theorem 8
For an MDP $\mathcal{M}$ *and a* δ*-shield, it holds for any state* s *that* $val^{\mathcal{M}}_s = val^{\mathcal{M}_{\text{U}}}_s$.

As optimal actions for the safety objective are not removed, optimality with respect to safety is preserved in the shielded MDP. Thus, during construction of the shield, we compute the action-valuations in fact *for the shielded MDP*. Observe that computing a shield for a state is done *independently* from the application of the shield to other states.

Guaranteed Safety. A δ-shield ensures that only actions that are δ-optimal with respect to an LTL property φ are allowed. In particular, for each action $a \in A_e$ at state s_e, we use the *minimal* probability $\eta^{\min}_{\varphi,\mathcal{M}}(s_e)$ to satisfy φ, see Definition 25. Under *optimal* (subsequent) choices, the value $\eta^{\min}_{\varphi,\mathcal{M}}(s_e)$ will be achieved. In contrast, a sequence of bad choices may violate φ with high probability. A more conservative notion would be to use the minimal action value while assuming that in all subsequent states the worst-case decisions corresponding to the maximal probabilities are taken. These values are computable by model checking [CHVB18]. Regardless of subsequent choices, at least $val^{\mathcal{M}}_{s_d}(a_e)$ is then guaranteed. A sensible notion to construct a shield would then be to impose a threshold $\lambda \in [0,1]$ such that only actions with $val^{\mathcal{M}}_{s_d}(a_e) \leq \lambda$ are allowed. A shield with such a guaranteed safety probability may induce a shielded MDP (Definition 27) that is *not deadlock free*. Moreover, the shield may become too restrictive for the agent.

Shielding versus Performance. A shield which is *minimal interfering* gives the RL agent the most freedom to optimize the performance objective. We propose two methods to alleviate interference, all of them assume *domain knowledge* of the rationale behind the decision procedure.

1. *Iterative Weakening.* During runtime, we may observe that the progress of the avatar regarding the performance objective is not increasing anymore.

In that case, we weaken the shield by $\delta - \varepsilon$, allowing additional actions. As soon as progress is made, we reset δ to its former value. The adaption of $\mathcal{S}^s_\delta$ to $\mathcal{S}^s_{\delta-\varepsilon}$ can be done on the fly, without new computations.

2. *Adapted Specifications.* If the goal of the decision maker is known *and* can be captured in temporal logic, we may adapt the original specification accordingly. There are often natural trade-offs between safety and performance. These trade-offs might be resolved via weights, but this process is often undesirable [RVWD13] and similar to reward engineering.

7.3.1 Scalable Shield Construction

Although we apply model checking only to the decision states in the safety-relevant MDP, scalability issues for large applications remain. We employ several optimizations towards computational tractability.

Finite horizon. For infinite horizon properties, the probability to violate safety (in the long run) is often one. Furthermore, our learned MDP model is inherently an approximation of the real world. Errors originating from this approximation accumulate for growing horizons. Thus, we focus on a finite horizon such that the action values (and consequently, a policy for the avatar) carry only guarantees for the next steps. This assumption also allows us to prune the safety-relevant MDP (see below), and thus increase the scalability of model checking.

Piecewise construction. Computing a shield for all states in an MDP concurrently yields a large memory footprint of the probabilistic model checker. To alleviate this footprint, we compute we can compute the shield states independently, in accordance with Theorem 8. The independent computation prunes the relevant part of the MDP, as the number of states reachable within the horizon is drastically reduced. Additionally, the independent computation allows for parallelizing the computation.

Independent agents. The explosion of state spaces stems mostly from the number of agents. Situationally, it may be possible to make the assumptions that the agents operate independently from each other. In particular, the probability for the avatar to crash with an adversary may be stochastically independent from crashing with the others. In such situations, instead of determining the shield for all adversaries at once, we perform computations for each agent individually, and combine them via the inclusion-exclusion principle. Afterward, the shield is composed from the shields dedicated to individual adversaries.

Abstractions. We observe that for finite horizon properties and piecewise construction, adversaries may be far away—beyond the horizon—without a chance to reach the avatar. We do not need to consider such (positions for) adversaries, as in these states, the shield will not block any actions.

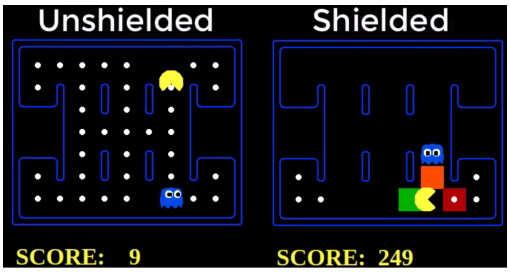


Figure 7.2: Still from video on small PacMan.

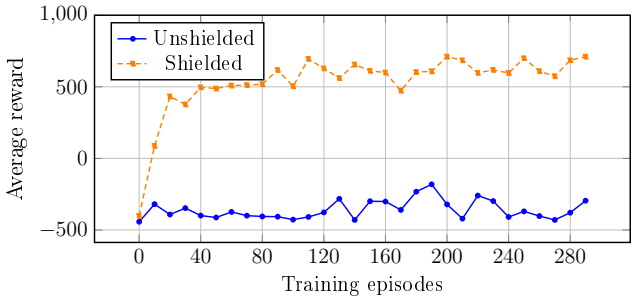


Figure 7.3: Scores during training for small PacMan.

7.4 Experimental Results

We run experiments using an Intel Core i7-4790K CPU with 16 GB of RAM using 4 cores. We give the timing results for a single CPU. Since the shield may be computed in a multi-threaded architecture, this time can be divided by the number of cores available. The supplementary materials, namely the source code and videos are available online.[2]

We demonstrate the applicability of our approach by means of two case studies: (1) the arcade game PacMan and (2) robots in a warehouse. For both case studies, we learn adversary behavior in small arenas individually for each adversary. These behavior models are applicable to any benchmark instance, as they are independent of concrete positions.

7.4.1 A shield for PacMan.

We extend the PacMan example from Section 6.6.2 from the deterministic setting to the probabilistic setting. For the deterministic setting, we assumed the ghost to be purely adversarial: for the construction of the deterministic shield, we did not consider any behavior model of the ghost and created a shield able to guarantee 100 percent safety for any possible movement of the ghost. Note, that it is possible to synthesize a deterministic shield for a single ghost only,

[2] `http://shieldrl.nilsjansen.org`

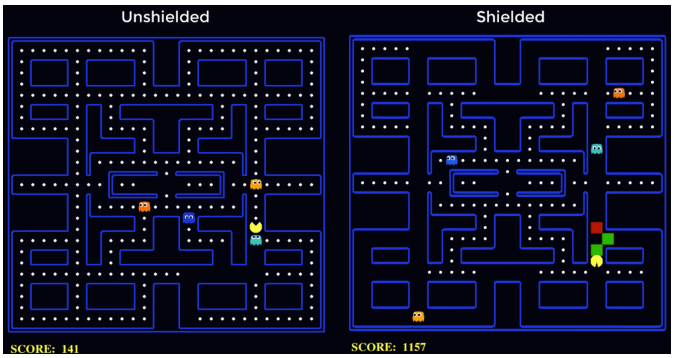


Figure 7.4: Still from video on classic PacMan.

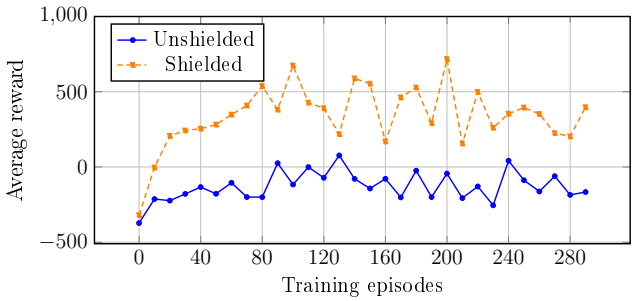


Figure 7.5: Scores during training for classic PacMan.

since two ghosts could always encircle PacMan. Therefore, no deterministic shield for more than one ghost exists. In this section, we synthesize probabilistic δ-shields for original versions of PacMan with up to four ghosts.

Setting. The setting is very similar to the setting in Section 6.6.2. We model various instance of the game (with different sizes) as an arena, where tokens represent the dots at each position in the maze, such that a dot is either present or collected. The score (reward, performance) is positively affected (+10) by collecting a dot and negatively by time (each step: -1). If PacMan either collects all dots (+500) or is caught (-500), the game is restarted.

The safety specification places a lower bound on the probability of reaching states in the underlying MDP that correspond to being caught.

Adversary models and safety-relevant MDP. Transferring the stochastic adversary behavior to any arena (without tokens) yields a concrete safety-relevant MDP. In particular, we specify an arena with the positions of the avatar and the adversaries as well as the behavior in the high-level `PRISM`-language [KNP11]. We employ a script that automatically generates arenas to enable a broad set of benchmarks. Taking, e.g., the PacMan arena from

Size, #Ghosts	#MC	time (s)	Score No Shield	Score Shield	Win Rate No Shield	Win Rate Shield
9x7,1	5912	584	-359,6	535,3	0,04	0,84
17x6,2	5841	1072	-195,6	253,9	0,04	0,4
17x10,3	51732	3681	-220,79	-40,52	0,01	0,07
27x25,4	269426	19941	-129,25	339,89	0,00	0,00

Table 7.1: Average scores and win rates for PacMan.

Figure 7.4, the considered MDP has roughly 10^{12} states (compared to 10^{50} for the full MDP). For a safety-relevant MDP, we compute a δ-shield (with iterative weakening) via the model checker `Storm` [DJKV17], using a horizon of 10 steps. The immense size even of safety-relevant MDPs requires optimizations such as a piecewise and independent shield construction, see Section 7.3.1. Moreover, a multi-threaded architecture lets us construct shields for very large examples. In particular, we perform model checking for (many) MDPs of roughly 10^6 states. The computation time for the largest PacMan instance takes about 6 hours, while memory is not an issue due to the piecewise shield construction.

Results. For our experiments, we reused the approximate Q-learning agent from Section 6.6.2 with the same feature vector and parameters. We compare RL to shielded RL on different instances. The key comparison criterion is the performance (detailed above) during learning. We describe results for the training phase of RL (300 episodes).

Figures 7.2 and 7.4 show screenshots of a series of videos. Each video compares how RL performs either shielded or unshielded on a instance of the case study. In the shielded version, at each decision state in the underlying MDP, we indicate the risk of decisions from low to high by the colors green, orange, red.

Figure 7.3 and Figure 7.5 depict the scores obtained during RL. The curves (blue, solid: unshielded, orange, dashed: shielded) show the average scores for every ten training episodes. Table 7.1 shows results for instances in increasing size. We list the number of model checking calls and the time to construct the shield. We list the scores with and without shield, and the *winning rate* capturing the ratio of successfully ended episodes. For all instances, we see a large difference in scores due to the fact that PacMan is often rescued by the shield. The winning rates differ for most benchmarks, favoring shielded RL. For three or four ghosts, a shield with a ten-step horizon cannot guide PacMan to avoid being encircled by the ghosts long enough to successfully end the game. Nevertheless, the shield often safes PacMan, leading to superior scores. Moreover, the shield helps learning an optimal policy much faster as fewer restarts are needed.

In general, learning for an arcade game like PacMan is difficult to perform according to safety constraints if no knowledge about future events is available. Given our relatively loose assumptions about the setting, the shield proved a

Figure 7.6: Still from the video on warehouse.

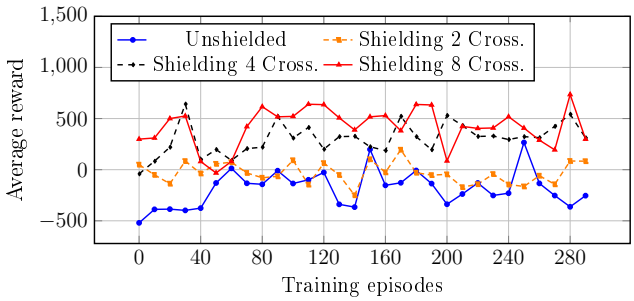


Figure 7.7: Scores during training for warehouse.

feasible means to ensure an appropriate measure of safety. Moreover, as in the classic PacMan instance, safety with certainty is not possible. In our experiments, we found that while finite horizons of 10 steps lead to good results, in several cases (such as long hallways), presumably safe behavior was in fact unsafe.

7.4.2 A Shield for Service Units in a Warehouse

As a second case study, we consider a warehouse floor plan with several corridors. A similar scenario has been investigated in [BLP+18]. In the arena, nodes describe crossings, the edges the corridors with shelves, and the distances the corridor length. The agents are fork-lift units picking up packages from the shelves and delivering them to the exit; tokens represent the presence of a package at its position. Corridors might be too narrow for multiple (facing) units, which poses a safety-critical situation. Most crucial is the crowded area near the exit, since all units have to deliver the packages to the exit.

Setting. The avatar is a specific (yellow) fork-lift unit that has to account for other units, the adversaries. The performance (reward) is positively affected by loading and delivering packages (+20, respectively) and negatively by time (each step: -1). Delivering all packages yields a large bonus (+500) and a collision leads

Crossings shielded	0	2	4	8
Score	-186	-27.6	303	420
Win Rate	0.16	0.31	0.59	0.71

Table 7.2: Average scores and win rates for warehouse.

to a large punishment (-500), both cases end the scenario.

Reinforcement Learning. Our implementation us4es an approximate Q-learning agent with the following feature vectors:

1. Has the unit loaded or unloaded,
2. the distance to the next package,
3. the distance to the exit,
4. whether another unit is three steps away, and
5. whether another unit is one step away.

The Q-learning uses the learning rate $\alpha = 0.2$ and the discount factor $\gamma = 0.8$ for the Q-update and an ϵ-greedy exploration policy with $\epsilon = 0.05$. We describe results for the training phase of RL (300 episodes).

Results. For the warehouse case study, we choose to vary the positions of the avatar for which we compute a shield. We present results for shielding the 2–8 crossings closest to the exit. Figures 7.6 shows a screenshot of a series of videos on the warehouse example. As for the PacMan example, each video compares how RL performs either shielded or unshielded on a instance of the case study. Figure 7.7 shows the average score for the different variants, and Table 7.2 summarizes average score and win rate. Unsurprisingly, the score gets better the more states are shielded. Furthermore, we have seen that shielding even more states has only a very limited effect.

Declaration of Sources

Chapter 7 was based on and reuses material from the following sources, previously published by the author:

- [JKJB18a] N. Jansen, B. Könighofer, S. Junges, R. Bloem: *Shielded decision-making in MDPs.* ArXiv. 2018

References to these sources are not always made explicit.

8 Conclusion

We believe that our work points to an exciting new direction for reactive synthesis in general and for applying synthesized controllers in real-world applications, because the set of critical properties of a complex system tends to be small and relatively easy to specify, thereby making shield synthesis scalable and usable. Many interesting extensions and variants remain to be explored, both theoretically and experimentally, in the future.

To conclude this thesis, we highlight the most important goals achieved and the contributions made to advance the prior state-of-the-art. Finally, we will present new challenges that emerged due to our work, and that remain for future investigation.

8.1 Summary and Goals Achieved

Shield synthesis is an approach to enforce safety properties at runtime. A shield monitors the system and corrects any erroneous output values instantaneously. Additionally, a shield deviates from the given outputs as little as it can.

In this thesis, we have formally defined the shield synthesis problem for reactive systems and presented a general framework for solving the problem. We distinguished between preemptive and post-posed shields and discussed basic synthesis approaches for both types of shields. Building on these basic shields, we considered shields with additional properties and applied different types of shields in various settings.

First, we considered shields with the ability to recover and to hand back control to the system as soon as possible. We discussed *k-stabilizing shields*, which guarantee recovery in a finite time, and *admissible shields*, which attempt to

work with the system to recover as soon as possible. Following, we discussed an extension of k-stabilizing shields, where erroneous output values of the reactive system are corrected while the liveness properties of the system are preserved.

Next, we considered shielding a human operator. A preemptive shield deactivates all actions that are unsafe in the current situation, and the human operator can choose from safe actions only. Additionally, the shield provided for every single deactivated action the information, which part of the specification was responsible for the deactivation. We called these shields *explanatory shields*.

Finally, we focused on shielding reinforcement learning agents. We developed a method for RL under safety constraints expressed as temporal logic specifications. The method is based on shielding the decisions of the underlying learning algorithm from violating the specification. We pursued two different approaches for shielded learning. First, we used deterministic post-posed and preemptive shields to guarantee safety with certainty during learning and execution. However, for some RL settings, this restriction on the decision-making of the learning agent might be to tight, and the agent may need to take some risks to explore the environment sufficiently. In such settings, we proposed probabilistic shields that incorporate more liberal constraints, that enforce safety violations to occur only with a small probability. Furthermore, in most experiments, the learning performance of the shielded agents improved compared to the unshielded case since unsafe policies do not have to be explored.

8.2 Future Work

Many ideas for new shield synthesis methods and new application areas for shields arose from working on shield synthesis for the last few years. We also came across several technical challenges along the way, for which further research is clearly indicated. We list a few possible directions for future work.

Shielding reinforcement learning agents. We see great potential for shields in its application in reinforcement learning. Recently, reinforcement learning, especially the state-of-art technology of deep reinforcement learning, has shown outstanding performance in a variety of tasks, and the list for successes of reinforcement learning is becoming longer almost by the day. The problem of safety in reinforcement is still an open research problem, which is attracting more attention every year. Shields proved a feasible means to ensure safety in the reinforcement learning setting. Since our work on shield synthesis was the first that combined formal synthesis techniques with machine learning and since this research area is so new, current approaches are not advanced enough for complex industrial use cases.

In future work, we will tackle the challenge of developing formal synthesis methods that can provide safety guarantees for modern learning algorithms, e.g., that employ deep recurrent neural networks as means of decision-making [HS15], applied in complex and changing environments.

Online shielding via model refinement and repair. The computation of shields is based on a faithful abstraction of the physical environment dynamics. In the case of an inaccurate model, the current approach fails, even for minor errors. We propose a self-correcting modeling approach. During runtime, a monitor observes the behavior of the environment. Whenever discrepancies between the model and the real environment dynamics are detected, the model will be corrected, and a new shield will be synthesized based on the updated model. This approach allows shielding in dynamic environments.

Shielding partially observable Markov decision processes. For future work, we will extend shields to richer models such as partially-observable MDPs (POMDPs). POMDPs are effective in modeling several real-world applications, e.g., modeling autonomous agents that make decisions under uncertainty and incomplete information. In this setting, while an agent makes decisions within an environment, it obtains observations and infers the likelihood of the system being in a particular state. The synthesis problem for POMDPs is to determine strategies that provably adhere to a probabilistic temporal logic specification. This problem is computationally intractable and theoretically hard [MHC99]. Shield synthesis could counteract scalability issues by considering only the safety part of the specification.

Shields for multi-agent systems. In [BBD+19], we proposed a general approach to the synthesis of shields for multi-agent systems. Swarm technology enables a large number of agents (e.g., UAVs) to become highly interconnected, with the ability to efficiently plan and allocate mission objectives, make coordinated tactical decisions, and collaboratively react to a dynamic environment with minimal supervision while making recommendations to human operators. This inter-connectivity with one another makes multi-agent systems a perfect setting for shielding to enforce global properties and to prevent congestions and collisions. In the paper, we demonstrated the applicability of the proposed shielding approach on a range of quantitative interference requirements by synthesizing shields for a multi-UAV system.

However, the current approach is not advanced and scalable enough to be applied on swarms with a large number of agents, and further research is needed to apply shields in advanced multi-agent settings. A promising avenue for future work is to investigate bounded synthesis with quantitative objectives to synthesize distributed shields.

Shields for timed systems. In future work, we will investigate methods to drive shields for real-time systems from timed automata specifications. We will consider shields for timed systems under different fault models, and we will extend our k-stabilizing shield synthesis approach to t-stabilizing shields: t-stabilizing shields guarantee recovery within t time units, regardless of future behavior of the system or the environment. First experiments implemented in the tool `Uppaal Tiga` suggest the potential of this new research direction.

Shields for cooperative systems. In previous work, we made no assumptions on the system to be shielded and treated it adversarially. Since the system might have bugs, modeling it as adversarial is reasonable. Though it is also a crude abstraction since typically, the objectives of the system and shield are similar. For future work, we plan to study ways to model the spectrum between cooperative and adversarial systems to be shielded together with solution concepts for the games that they give rise to.

8.3 Last Words

I conclude this thesis with the following words:

"One tool to control them all, One tool to save them,
One tool to shield them all, and through their failures guide them."

by Florian Lorber

Bibliography

[ABC+19] Guy Avni, Roderick Bloem, Krishnendu Chatterjee, Thomas A. Henzinger, Bettina Könighofer, and Stefan Pranger. Run-time optimization for learned controllers through quantitative games. In *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I*, pages 630–649, 2019.

[ABE+17] Mohammed Alshiekh, Roderick Bloem, Rüdiger Ehlers, Bettina Könighofer, Scott Niekum, and Ufuk Topcu. Safe reinforcement learning via shielding. *CoRR*, abs/1708.08611, 2017.

[ABE+18] Mohammed Alshiekh, Roderick Bloem, Rüdiger Ehlers, Bettina Könighofer, Scott Niekum, and Ufuk Topcu. Safe reinforcement learning via shielding. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, pages 2669–2678, 2018.

[AKL+19] Pranav Ashok, Jan Kretínský, Kim Guldstrand Larsen, Adrien Le Coënt, Jakob Haahr Taankvist, and Maximilian Weininger. SOS: safe, optimal and small strategies for hybrid markov decision processes. In *Quantitative Evaluation of Systems, 16th International Conference, QEST 2019, Glasgow, UK, September 10-12, 2019, Proceedings*, pages 147–164, 2019.

[AOS+16] Dario Amodei, Chris Olah, Jacob Steinhardt, Paul Christiano, John Schulman, and Dan Mané. Concrete problems in AI safety. *CoRR*, abs/1606.06565, 2016.

[AS85] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, October 1985.

[Bas19] Osbert Bastani. Safe reinforcement learning via online shielding. *CoRR*, abs/1905.10691, 2019.

[BBD+19] Suda Bharadwaj, Roderik Bloem, Rayna Dimitrova, Bettina Könighofer, and Ufuk Topcu. Synthesis of minimum-cost shields for multi-agent systems. In *2019 American Control Conference, ACC 2019, Philadelphia, PA, USA, July 10-12, 2019*, pages 1048–1055, 2019.

[BCC+14] Tomáš Brázdil, Krishnendu Chatterjee, Martin Chmelík, Vojtěch Forejt, Jan Křetínský, Marta Z. Kwiatkowska, David Parker, and Mateusz Ujma. Verification of Markov decision processes using learning algorithms. In *ATVA*, 2014.

[BCG+10] Roderick Bloem, Krishnendu Chatterjee, Karin Greimel, Thomas A. Henzinger, and Barbara Jobstmann. Robustness in the presence of liveness. In *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, pages 410–424, 2010.

[BCG+14] Roderick Bloem, Krishnendu Chatterjee, Karin Greimel, Thomas A. Henzinger, Georg Hofferek, Barbara Jobstmann, Bettina Könighofer, and Robert Könighofer. Synthesizing robust systems. *Acta Inf.*, 51(3-4):193–220, 2014.

[BCN+19] Suda Bharadwaj, Steven Carr, Natasha Neogi, Hasan Poonawala, Alejandro Barberia Chueca, and Ufuk Topcu. Traffic management for urban air mobility. In *NASA Formal Methods - 11th International Symposium, NFM 2019, Houston, TX, USA, May 7-9, 2019, Proceedings*, pages 71–87, 2019.

[Ber18] UC Berkeley. Intro to AI – Reinforcement Learning , 2018. http://ai.berkeley.edu/reinforcement.html.

[BGH+12] Roderick Bloem, Hans-Jürgen Gamauf, Georg Hofferek, Bettina Könighofer, and Robert Könighofer. Synthesizing robust systems with RATSY. In *Proceedings First Workshop on Synthesis, SYNT 2012, Berkeley, California, USA, 7th and 8th July 2012.*, pages 47–53, 2012.

[BGI+18] Roderick Bloem, Hannes Groß, Rinat Iusupov, Bettina Könighofer, Stefan Mangard, and Johannes Winter. Formal verification of masked hardware implementations in the presence of glitches. In *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part II*, pages 321–353, 2018.

[BGJ+07] Roderick Bloem, Stefan J. Galler, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Martin Weiglhofer. Automatic hardware synthesis from specifications: a case study. In *DATE*, pages 1188–1193, 2007.

[BHK+14] Roderick Bloem, Georg Hofferek, Bettina Könighofer, Robert Könighofer, Simon Ausserlechner, and Raphael Spork. Synthesis of synchronization using uninterpreted functions. In *Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, October 21-24, 2014*, pages 35–42, 2014.

[BJP+12] R. Bloem, B. Jobstmann, N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of reactive(1) designs. *J. Comput. Syst. Sci.*, 78(3):911–938, 2012.

[BJW02] Julien Bernet, David Janin, and Igor Walukiewicz. Permissive strategies: From parity games to safety games. *ITA*, 36(3):261–275, 2002.

[BK08] Christel Baier and J.P. Katoen. *Principles of Model Checking*. MIT Press, 2008.

[BKKW15] Roderick Bloem, Bettina Könighofer, Robert Könighofer, and Chao Wang. Shield synthesis: - runtime enforcement for reactive systems. In *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, pages 533–548, 2015.

[BKN+19] Maxime Bouton, Jesper Karlsson, Alireza Nakhaei, Kikuo Fujimura, Mykel J. Kochenderfer, and Jana Tumova. Reinforcement learning with probabilistic guarantees for autonomous driving. *CoRR*, abs/1904.07189, 2019.

[BL69] J. Richard Büchi and Lawrence H. Landweber. Solving sequential conditions by finite-state strategies. *Transactions of the American Mathematical Society*, 138:295–311, 1969.

[BLP+18] Arthur Bit-Monnot, Francesco Leofante, Luca Pulina, Erika Ábrahám, and Armando Tacchella. Smartplan: a task planner for smart factories. *CoRR*, abs/1806.07135, 2018.

[BLS11] Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime verification for LTL and TLTL. *ACM Trans. Softw. Eng. Methodol.*, 20(4):14:1–14:64, 2011.

[BM10] R. K. Brayton and A. Mishchenko. ABC: An academic industrial-strength verification tool. In *CAV*, LNCS 6174, pages 24–40. Springer, 2010.

[Bry86] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Computers*, 35(8):677–691, 1986.

[BT96] Ronen I Brafman and Moshe Tennenholtz. On partially controlled multi-agent systems. *J. of Artif. Intell. Res.*, 4:477–507, 1996.

[BTSK17] Felix Berkenkamp, Matteo Turchetta, Angela Schoellig, and Andreas Krause. Safe model-based reinforcement learning with stability guarantees. In *NIPS*, pages 908–919, 2017.

[CCC10] H. Chao, Y. Cao, and Y. Chen. Autopilots for small unmanned aerial vehicles: A survey. *Int. J. Control, Automation and Systems*, 8(1):36 – 44, 2010.

[CE81] Edmund M Clarke and E Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on Logic of Programs*, pages 52–71. Springer, 1981.

[Chu63] Alonzo Church. Logic, arithmetic, and automata. In *Proceedings of the International Congress of Mathematicians (ICM'62)*, pages 23–35, 1963.

[CHVB18] Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors. *Handbook of Model Checking*. Springer, 2018.

[Clo97] J. Clouse. On integrating apprentice learning and reinforcement learning title2:. Technical report, Amherst, MA, USA, 1997.

[CMH06] Krishnendu Chatterjee, Rupak Majumdar, and Thomas A. Henzinger. Markov Decision Processes with Multiple Objectives. In *STACS*, volume 3884 of *LNCS*, pages 325–336. Springer, 2006.

[CNDGG18] Y. Chow, O. Nachum, E. Duenez-Guzman, and M. Ghavamzadeh. A Lyapunov-based approach to safe reinforcement learning. In *NIPS*, pages 8103–8112, 2018.

[CRST08] Alessandro Cimatti, Marco Roveri, Viktor Schuppan, and Andrei Tchaltsev. Diagnostic information for realizability. In *Verification, Model Checking, and Abstract Interpretation, 9th International Conference, VMCAI 2008, San Francisco, USA, January 7-9, 2008, Proceedings*, pages 52–67, 2008.

[DAC99] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *ICSE*, pages 411–420. ACM, 1999.

[DGS+19] Ankush Desai, Shromona Ghosh, Sanjit A. Seshia, Natarajan Shankar, and Ashish Tiwari. SOTER: A runtime assurance framework for programming safe robotics systems. In *49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2019, Portland, OR, USA, June 24-27, 2019*, pages 138–150, 2019.

[Die00] Thomas G Dietterich. Hierarchical reinforcement learning with the maxq value function decomposition. *Journal of Artificial Intelligence Research*, 13:227–303, 2000.

[DJKV17] Christian Dehnert, Sebastian Junges, Joost-Pieter Katoen, and Matthias Volk. A storm is coming: A modern probabilistic model checker. In *CAV (2)*, volume 10427 of *LNCS*, pages 592–600. Springer, 2017.

[DJL+15] Alexandre David, Peter Gjøl Jensen, Kim Guldstrand Larsen, Marius Mikucionis, and Jakob Haahr Taankvist. Uppaal stratego. In *TACAS*, volume 9035 of *Lecture Notes in Computer Science*, pages 206–211. Springer, 2015.

[DN08] Peter Dayan and Yael Niv. Reinforcement learning: the good, the bad and the ugly. *Current opinion in neurobiology*, 18(2):185–196, 2008.

[DNC10] B. Donmez, C. Nehme, and M. L. Cummings. Modeling workload impact in multiple unmanned vehicle supervisory control. *IEEE Trans. Syst. Man Cybern. A., Syst. Humans*, 40(6):1180 – 1190, 2010.

[Duq09] M. Duquette. Effects-level models for UAV simulation. In *AIAA Modeling and Simulation Technologies Conference*, 2009.

[DVP11] K. Dalamagkidis, K. P. Valavanis, and L. A. Piegl. *On integrating unmanned aircraft systems into the national airspace system: Issues, challenges, operational restrictions, certification, and recommendations*, volume 54. Springer Science & Business Media, 2011.

[EKH12] Rüdiger Ehlers, Robert Könighofer, and Georg Hofferek. Symbolically synthesizing small circuits. In *Formal Methods in Computer-Aided Design, FMCAD 2012, Cambridge, UK, October 22-25, 2012*, pages 91–100, 2012.

[ER16] Rüdiger Ehlers and Vasumathi Raman. Slugs: Extensible GR(1) synthesis. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, pages 333–339, 2016.

[ET14] R. Ehlers and U. Topcu. Resilience to intermittent assumption violations in reactive synthesis. In *17th I. Conference on Hybrid Systems: Computation and Control, HSCC'14, Berlin, Germany, April 15-17, 2014*, pages 203–212, 2014.

[ET15] Rüdiger Ehlers and Ufuk Topcu. Estimator-based reactive synthesis under incomplete information. In *Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control, HSCC'15, Seattle, WA, USA, April 14-16, 2015*, pages 249–258, 2015.

[Fae09] M. Faella. Admissible strategies in infinite games over graphs. In *Mathematical Foundations of Computer Science 2009, 34th I. Symposium, MFCS 2009, Novy Smokovec, Slovakia, 2009*, pages 307–318, 2009.

[Fal10] Yliès Falcone. You should better enforce than verify. In *Runtime Verification - First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings*, pages 89–105, 2010.

[FFM12] Y. Falcone, J.-C. Fernandez, and L. Mounier. What can you verify and enforce at runtime? *STTT*, 14(3):349–382, 2012.

[FJR09] Emmanuel Filiot, Naiyong Jin, and Jean-François Raskin. An antichain algorithm for LTL realizability. In *CAV*, pages 263–277, 2009.

[FP18] Nathan Fulton and André Platzer. Safe reinforcement learning via formal methods: Toward safe control through proof and learning. In *AAAI*. AAAI Press, 2018.

[Ful18] Nathan Fulton. Verifiably safe autonomy for cyber-physical systems. Ph. D. thesis, Computer Science Department, School of Computer Science, Carnegie Mellon University, 2018.

[FWHT16] L. Feng, Clemens Wiltsche, L. Humphrey, and U. Topcu. Synthesis of human-in-the-loop control protocols for autonomous systems. In *IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*, 2016.

[FZ16] Richard G Freedman and Shlomo Zilberstein. Safety in AI-HRI: Challenges complementing user experience quality. In *AAAI Fall Symposium Series*, 2016.

[GF15] Javier Garcıa and Fernando Fernández. A comprehensive survey on safe reinforcement learning. *Journal of Machine Learning Research*, 16(1):1437–1480, 2015.

[HAK18] Mohammadhosein Hasanbeig, Alessandro Abate, and Daniel Kroening. Logically-correct reinforcement learning. *CoRR*, abs/1801.08099, 2018.

[HGK+13] Georg Hofferek, Ashutosh Gupta, Bettina Könighofer, Jie-Hong Roland Jiang, and Roderick Bloem. Synthesizing multiple boolean functions using interpolation on a single proof. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 77–84, 2013.

[HH14] Arnd Hartmanns and Holger Hermanns. The modest toolset: An integrated environment for quantitative modelling and verification. In *TACAS*, volume 8413 of *LNCS*, pages 593–598, 2014.

[HHH+19] Ernst Moritz Hahn, Arnd Hartmanns, Christian Hensel, Michaela Klauck, Joachim Klein, Jan Kretínský, David Parker, Tim Quatmann, Enno Ruijters, and Marcel Steinmetz. The 2019 comparison of tools for the analysis of quantitative formal models - (qcomp 2019 competition report). In *TACAS (3)*, volume 11429 of *Lecture Notes in Computer Science*, pages 69–92. Springer, 2019.

[HK99] Thomas A. Henzinger and Peter W. Kopke. Discrete-time control for rectangular hybrid automata. *Theor. Comput. Sci.*, 221(1-2):369–392, 1999.

[HKKT16] Laura R. Humphrey, Bettina Könighofer, Robert Könighofer, and Ufuk Topcu. Synthesis of admissible shields. In *Hardware and Software: Verification and Testing - 12th International Haifa Verification Conference, HVC 2016, Haifa, Israel, November 14-17, 2016, Proceedings*, pages 134–151, 2016.

[HPS+19] Ernst Moritz Hahn, Mateo Perez, Sven Schewe, Fabio Somenzi, Ashutosh Trivedi, and Dominik Wojtczak. Omega-regular objectives in model-free reinforcement learning. In *TACAS (1)*, volume 11427 of *Lecture Notes in Computer Science*, pages 395–412. Springer, 2019.

[HS15] Matthew Hausknecht and Peter Stone. Deep recurrent q-learning for partially observable mdps. *CoRR, abs/1507.06527*, 2015.

[JB12] Swen Jacobs and Roderick Bloem. Parameterized synthesis. In *TACAS*, pages 362–376, 2012.

[JGWB07] Barbara Jobstmann, Stefan J. Galler, Martin Weiglhofer, and Roderick Bloem. Anzu: A tool for property synthesis. In *CAV*, pages 258–262, 2007.

[JKJB18a] Nils Jansen, Bettina Könighofer, Sebastian Junges, and Roderick Bloem. Shielded decision-making in mdps. *CoRR*, abs/1807.06096, 2018.

[JKJB18b] Nils Jansen, Bettina Könighofer, Sebastian Junges, and Roderick Bloem. Shielded decision-making in mdps. *CoRR*, abs/1807.06096, 2018.

[JND+16] Sebastian Junges, Nils Jansen, Christian Dehnert, Ufuk Topcu, and Joost-Pieter Katoen. Safety-constrained reinforcement learning for MDPs. In *TACAS*, volume 9636 of *LNCS*, pages 130–146. Springer, 2016.

[KAB+17] Bettina Könighofer, Mohammed Alshiekh, Roderick Bloem, Laura R. Humphrey, Robert Könighofer, Ufuk Topcu, and Chao Wang. Shield synthesis. *Formal Methods in System Design*, 51(2):332–361, 2017.

[Kat16] Joost-Pieter Katoen. The probabilistic model checking landscape. In *LICS*, pages 31–45. ACM, 2016.

[KHB13] Robert Könighofer, Georg Hofferek, and Roderick Bloem. Debugging formal specifications: A practical approach using model-based diagnosis and counterstrategies. *STTT*, 15(5-6):563–583, 2013.

[KNP11] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *CAV*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011.

[Kol12] Andrey Kolobov. Planning with Markov decision processes: An AI perspective. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 6(1):1–210, 2012.

[KPR18] Jan Kretínský, Guillermo A. Pérez, and Jean-François Raskin. Learning-based mean-payoff optimization in an unknown MDP under omega-regular constraints. In *CONCUR*, volume 118 of *LIPIcs*, pages 8:1–8:18. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *NIPS*, pages 1097–1105, 2012.

[Kwi03] Marta Z. Kwiatkowska. Model checking for probability and time: from theory to practice. In *LICS*, page 351. IEEE CS, 2003.

[LBR09] R. Loh, Y. Bian, and T. Roe. UAVs in civil airspace: Safety requirements. *IEEE Aerospace and Electronic Systems Magazine*, 24(1):5–17, Jan 2009.

[LBW09] J. Ligatti, L. Bauer, and D. Walker. Run-time enforcement of nonsafety policies. *ACM Trans. Inf. Syst. Secur.*, 12(3), 2009.

[LS08] Mark H. Liffiton and Karem A. Sakallah. Algorithms for computing minimal unsatisfiable subsets of constraints. *J. Autom. Reasoning*, 40(1):1–33, 2008.

[LS09] M. Leucker and S. Schallhart. A brief account of runtime verification. *The Journal of Logic and Algebraic Programming*, 78(5):293–303, 2009.

[LSSS14] W. Li, D. Sadigh, S. Sastry, and S. Seshia. Synthesis for human-in-the-loop control systems. In *Tools and Algorithms for the Construction and Analysis of Systems - 20th I. Conference, TACAS 2014, Grenoble, France, April 5-13, 2014. Proceedings*, pages 470–484, 2014.

[LTL] *LTL Specification Patterns.* `http://patterns.projects.cis.ksu.edu/documentation/patterns/ltl.shtml`.

[MA12] Teodor Mihai Moldovan and Pieter Abbeel. Safe exploration in markov decision processes. In *ICML*. icml.cc / Omnipress, 2012.

[Maz01] R. Mazala. Infinite games. In *Automata, Logics, and Infinite Games: A Guide to Current Research*, LNCS 2500, pages 23–42. Springer, 2001.

[MCKB17] George Mason, Radu Calinescu, Daniel Kudenko, and Alec Banks. Assured reinforcement learning with formally verified abstract policies. In *ICAART (2)*, pages 105–117. SciTePress, 2017.

[MHC99] Omid Madani, Steve Hanks, and Anne Condon. On the undecidability of probabilistic planning and infinite-horizon partially observable markov decision problems. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence and Eleventh Conference on Innovative Applications of Artificial Intelligence, July 18-22, 1999, Orlando, Florida, USA.*, pages 541–548, 1999.

[MMM+14] T. Mancini, F. Mari, A. Massini, I. Melatti, and E. Tronci. Anytime system level verification via random exhaustive hardware in the loop simulation. In *Digital System Design (DSD), 2014 17th Euromicro Conference on*, pages 236–245, Aug 2014.

[Nat16] National Science and Technology Council. *Preparing for the Future of Artificial Intelligence*. 2016.

[OWNE19] M. Ohnishi, L. Wang, G. Notomista, and M. Egerstedt. Barrier-certified adaptive reinforcement learning with applications to brushbot navigation. *IEEE Transactions on Robotics*, pages 1–20, 2019.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pages 46–57, 1977.

[PP06] Nir Piterman and Amir Pnueli. Faster solutions of rabin and streett games. In *21th IEEE Symposium on Logic in Computer Science (LICS 2006), 12-15 August 2006, Seattle, WA, USA, Proceedings*, pages 275–284, 2006.

[PR89] A. Pnueli and R. Rosner. *Automata, Languages and Programming: 16th Int. Colloquium Stresa, Italy, July 11–15, 1989 Proceedings*, chapter On the synthesis of an asynchronous reactive module, pages 652–671. Springer Berlin Heidelberg, Berlin, Heidelberg, 1989.

[PS14] Martin Pecka and Tomas Svoboda. Safe exploration techniques for reinforcement learning–an overview. In *International Workshop on Modelling and Simulation for Autonomous Systems*, pages 357–375. Springer, 2014.

[Put94] Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1994.

[PYG+17] Dung Phan, Junxing Yang, Radu Grosu, Scott A. Smolka, and Scott D. Stoller. Collision avoidance for mobile robots with limited sensing and limited information about moving obstacles. *Formal Methods in System Design*, 51(1):62–86, 2017.

[QS82] J. P. Quielle and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Symposium on Programming*, LNCS 137. Springer, 1982.

[RDT16] Stuart J. Russell, Daniel Dewey, and Max Tegmark. Research priorities for robust and beneficial artificial intelligence. *CoRR*, abs/1602.03506, 2016.

[Ros92] R. Rosner. *Modular Synthesis of Reactive Systemsk*. PhD thesis, Weizmann Institute of Science, Rehovot, Israel, 1992.

[RVWD13] Diederik M. Roijers, Peter Vamplew, Shimon Whiteson, and Richard Dazeley. A survey of multi-objective sequential decision-making. *J. Artif. Intell. Res.*, 48:67–113, 2013.

[SB98] Richard S Sutton and Andrew G Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[Sch00] F. B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3:30–50, 2000.

[SF07] Sven Schewe and Bernd Finkbeiner. Bounded synthesis. In *ATVA*, pages 474–488, 2007.

[Sha01] Lui Sha. Using simplicity to control complexity. *IEEE Software*, 18(4):20–28, 2001.

[SHB16] Marcel Steinmetz, Jörg Hoffmann, and Olivier Buffet. Goal probability analysis in probabilistic planning: Exploring and enhancing the state of the art. *J. Artif. Intell. Res.*, 57:229–271, 2016.

[SKC+14] Dorsa Sadigh, Eric S Kim, Samuel Coogan, S Shankar Sastry, and Sanjit A Seshia. A learning based approach to control synthesis of markov decision processes for linear temporal logic specifications. In *Decision and Control (CDC), 2014 IEEE 53rd Annual Conference on*, pages 1091–1096. IEEE, 2014.

[SLS+18] Dorsa Sadigh, Nick Landolfi, Shankar S Sastry, Sanjit A Seshia, and Anca D Dragan. Planning for cars that coordinate with people: leveraging effects on human actions for planning and active information gathering over human internal state. *Autonomous Robots*, 42(7):1405–1426, 2018.

[SPE+14] D Sculley, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, and Michael Young. Machine learning: The high-interest credit card of technical debt. 2014.

[SS13] Saqib Sohail and Fabio Somenzi. Safety first: a two-stage algorithm for the synthesis of reactive systems. *STTT*, 15(5-6):433–454, 2013.

[SSP+17] Ion Stoica, Dawn Song, Raluca Ada Popa, David Patterson, Michael W Mahoney, Randy Katz, Anthony D Joseph, Michael Jordan, Joseph M Hellerstein, Joseph E Gonzalez, et al. A berkeley view of systems challenges for AI. *CoRR*, abs/1712.05855, 2017.

[SSSD16] Dorsa Sadigh, Shankar Sastry, Sanjit A Seshia, and Anca D Dragan. Planning for autonomous cars that leverage effects on human actions. In *Robotics: Science and Systems*, 2016.

[TB06] Andrea L. Thomaz and Cynthia Breazeal. Reinforcement learning with human teachers: Evidence of feedback and guidance with implications for learning performance. In *Proceedings of the 21st National Conference on Artificial Intelligence - Vol. 1*, pages 1000–1005. AAAI Press, 2006.

[TB08] Andrea Lockerd Thomaz and Cynthia Breazeal. Teachable robots: Understanding human teaching behavior to build more effective robot learners. *Artif. Intell.*, 172(6-7):716–737, 2008.

[Tho95] Wolfgang Thomas. On the synthesis of strategies in infinite games. In *STACS 95, 12th Annual Symposium on Theoretical Aspects of Computer Science, Munich, Germany, March 2-4, 1995, Proceedings*, pages 1–13, 1995.

[VdHW08] Wiebe Van der Hoek and Michael Wooldridge. Multi-agent systems. *Foundations of Artificial Intelligence*, 3:887–928, 2008.

[WET15] Min Wen, Rüdiger Ehlers, and Ufuk Topcu. Correct-by-synthesis reinforcement learning with temporal logic constraints. In *IROS*, pages 4983–4990. IEEE CS, 2015.

[WFHP16] Ian H Witten, Eibe Frank, Mark A Hall, and Christopher J Pal. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2016.

[Whi85] Douglas J White. Real applications of Markov decision processes. *Interfaces*, 15(6):73–83, 1985.

[WTM12] Tichakorn Wongpiromsarn, Ufuk Topcu, and Richard M. Murray. Receding horizon temporal logic planning. *IEEE Trans. Automat. Contr.*, 57(11):2817–2830, 2012.

[WWDW19] Meng Wu, Jingbo Wang, Jyotirmoy Deshmukh, and Chao Wang. Shield synthesis for real: Enforcing safety in cyber-physical systems, 2019.

[ZMBD08] Brian D. Ziebart, Andrew L. Maas, J. Andrew Bagnell, and Anind K. Dey. Maximum entropy inverse reinforcement learning. In *AAAI*, pages 1433–1438. AAAI Press, 2008.

[ZXMJ19] He Zhu, Zikang Xiong, Stephen Magill, and Suresh Jagannathan. An inductive synthesis framework for verifiable reinforcement learning. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019.*, pages 686–701, 2019.

A List of Publications

According to the Statutes of the Doctoral School of Computer Science at Graz University of Technology, a PhD thesis must contain a list of publications of the candidate, detailing the relationship between the thesis and the (relevant) publications. The reference date for the following list of publications is December 2019.

A.1 Journal Publications

This section lists all journal publications in reverse chronological order.

- [KAB+17] Bettina Könighofer, Mohammed Alshiekh, Roderick Bloem, Laura R. Humphrey, Robert Könighofer, Ufuk Topcu, and Chao Wang. Shield synthesis. *Formal Methods in System Design*, 51(2):332–361, 2017.

- [BCG+14] Roderick Bloem, Krishnendu Chatterjee, Karin Greimel, Thomas A. Henzinger, Georg Hofferek, Barbara Jobstmann, Bettina Könighofer, and Robert Könighofer. Synthesizing robust systems. *Acta Inf.*, 51(3-4):193–220, 2014.

A.2 Publications in Conference and Workshop Proceedings

This section lists all publications in conference and workshop proceedings in reverse chronological order.

- [BBD+19] Suda Bharadwaj, Roderik Bloem, Rayna Dimitrova, Bettina Könighofer, and Ufuk Topcu. Synthesis of minimum-cost shields for multi-agent systems. In *2019 American Control Conference, ACC 2019, Philadelphia, PA, USA, July 10-12, 2019*, pages 1048–1055, 2019

- [ABC+19] Guy Avni, Roderick Bloem, Krishnendu Chatterjee, Thomas A. Henzinger, Bettina Könighofer, and Stefan Pranger. Run-time optimization for learned controllers through quantitative games. In *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I*, pages 630–649, 2019

- [ABE+18] Mohammed Alshiekh, Roderick Bloem, Rüdiger Ehlers, Bettina Könighofer, Scott Niekum, and Ufuk Topcu. Safe reinforcement learning via shielding. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, pages 2669–2678, 2018

- [BGI+18] Roderick Bloem, Hannes Groß, Rinat Iusupov, Bettina Könighofer, Stefan Mangard, and Johannes Winter. Formal verification of masked hardware implementations in the presence of glitches. In *Advances in Cryptology - EUROCRYPT 2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part II*, pages 321–353, 2018

- [HKKT16] Laura R. Humphrey, Bettina Könighofer, Robert Könighofer, and Ufuk Topcu. Synthesis of admissible shields. In *Hardware and Software: Verification and Testing - 12th International Haifa Verification Conference, HVC 2016, Haifa, Israel, November 14-17, 2016, Proceedings*, pages 134–151, 2016

- [BKKW15] Roderick Bloem, Bettina Könighofer, Robert Könighofer, and Chao Wang. Shield synthesis: - runtime enforcement for reactive systems. In *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, pages 533–548, 2015

- [BHK+14] Roderick Bloem, Georg Hofferek, Bettina Könighofer, Robert Könighofer, Simon Ausserlechner, and Raphael Spork. Synthesis of synchronization using uninterpreted functions. In *Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, October 21-24, 2014*, pages 35–42, 2014

- [HGK+13] Georg Hofferek, Ashutosh Gupta, Bettina Könighofer, Jie-Hong Roland Jiang, and Roderick Bloem. Synthesizing multiple boolean functions using interpolation on a single proof. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 77–84, 2013

- [BGH+12] Roderick Bloem, Hans-Jürgen Gamauf, Georg Hofferek, Bettina Könighofer, and Robert Könighofer. Synthesizing robust systems with RATSY. In *Proceedings First Workshop on Synthesis, SYNT 2012, Berkeley, California, USA, 7th and 8th July 2012.*, pages 47–53, 2012

A.3 Informal Publications

This section lists all informal publications in reverse chronological order.

- [JKJB18b] Nils Jansen, Bettina Könighofer, Sebastian Junges, and Roderick Bloem. Shielded decision-making in mdps. *CoRR*, abs/1807.06096, 2018

- [ABE+17] Mohammed Alshiekh, Roderick Bloem, Rüdiger Ehlers, Bettina Könighofer, Scott Niekum, and Ufuk Topcu. Safe reinforcement learning via shielding. *CoRR*, abs/1708.08611, 2017

A.4 Relationship between Publications and Thesis

This thesis is based on the following publications (in reverse chronological order):

- [ABE+18] Mohammed Alshiekh, Roderick Bloem, Rüdiger Ehlers, Bettina Könighofer, Scott Niekum, and Ufuk Topcu. Safe reinforcement learning via shielding. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, pages 2669–2678, 2018

- [JKJB18b] Nils Jansen, Bettina Könighofer, Sebastian Junges, and Roderick Bloem. Shielded decision-making in mdps. *CoRR*, abs/1807.06096, 2018

- [KAB+17] Bettina Könighofer, Mohammed Alshiekh, Roderick Bloem, Laura R. Humphrey, Robert Könighofer, Ufuk Topcu, and Chao Wang. Shield synthesis. *Formal Methods in System Design*, 51(2):332–361, 2017

- [ABE+17] Mohammed Alshiekh, Roderick Bloem, Rüdiger Ehlers, Bettina Könighofer, Scott Niekum, and Ufuk Topcu. Safe reinforcement learning via shielding. *CoRR*, abs/1708.08611, 2017

- [HKKT16] Laura R. Humphrey, Bettina Könighofer, Robert Könighofer, and Ufuk Topcu. Synthesis of admissible shields. In *Hardware and Software: Verification and Testing - 12th International Haifa Verification Conference, HVC 2016, Haifa, Israel, November 14-17, 2016, Proceedings*, pages 134–151, 2016

- [BKKW15] Roderick Bloem, Bettina Könighofer, Robert Könighofer, and Chao Wang. Shield synthesis: - runtime enforcement for reactive systems. In *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, pages 533–548, 2015

Throughout this thesis, sections that reuse material from these publications are marked with grey boxes entitled: *Declaration of Sources.* These boxes are placed at the end of each chapter. In addition to that, the most important relations between these publications and this dissertation are stated below.

In [BKKW15], we introduced the concept of a shield and defined a general framework for solving the shield synthesis problem for reactive systems. Thus, Chapter 1 and Section 3.2 are largely based on this paper. Furthermore, we proposed in [BKKW15] k-stabilizing shields which we discuss in Section 4.2.

In [HKKT16], we proposed another synthesis method to automatically construct shields for reactive systems and called the resulting shields admissible shields. In this thesis, admissible shields are discussed in Section 4.3.

In [KAB+17], we summarized the approaches of [BKKW15] and [HKKT16]. Additionally, the paper introduced liveness-preserving shielding which is the basis for Section 4.4, and introduced explanatory shields which is the basis for Chapter 5.

In [ABE+18], we discussed post-posed shields for reinforcement learning agents in the deterministic setting. Thus, Section 6.3 is largely based on [ABE+18].

The informal publication [ABE+17] is an extended version of [ABE+18], which proposed preemptive shields for reinforcement learning agents. Thus, Section 6.4 is largely based on [ABE+17].

In [JKJB18b], we discussed shields for reinforcement learning agents in the probabilistic setting. This publication is the basis for Chapter 7.

B
Cooperations

According to the Statutes of the Doctoral School of Computer Science at Graz University of Technology, a PhD thesis must contain an explanation of cooperations concerning the work described in the thesis. The following list details all such notable cooperations between the author and other persons. The list is in no particular order.

- On all parts of this thesis, there have been frequent and ongoing discussions with Roderick Bloem.
- Ufuk Topcu participated in many discussions on many different aspects of this thesis and provided valuable feedback.
- The initial idea of shields was developed in cooperation with Chao Wang.
- The concept of shields for reinforcement learning was developed in close cooperation with Rüdiger Ehlers.
- Scott Niekum provided his vast expertise in machine learning which enabled us to formulate the shielding concept for the AI community proper.
- Laura R. Humphrey contributed to many fruitful discussions on suitable application domains for shields, which resulted in a detailed case study on shields for UAV mission planning.
- Mohammed Alshiekh assisted in performing the case study on shields for UAV mission planning.
- Robert Könighofer contributed to the initial developments of the shielding concept by helping to develop the theory for k-stabilizing shields and by assisting with the experiments.

- The concept of shielded decision making for MDPs was developed in cooperation with Nils Jansen and Sebastian Junges.

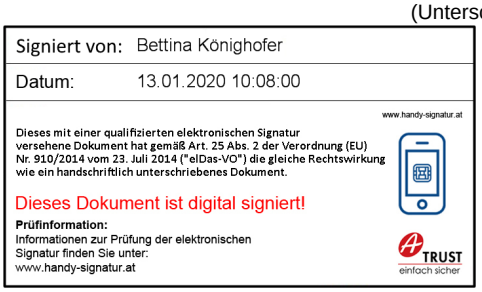
Deutsche Fassung:
Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008
Genehmigung des Senates am 1.12.2008

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Graz, am …… ……..
(Unterschrift)

Englische Fassung:



STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

…… ……..
date (signature)